COMPILER DIRECTED CODESIGN FOR FPGA-BASED EMBEDDED SYSTEMS

A thesis submitted in fulfilment of the requirements

for the degree of Doctor of Philosophy

Martin A. Hauff

B.Eng. (Computer Systems)

School of Electrical and Computer Engineering

Science, Engineering and Technology Portfolio

RMIT University

October 2007

DECLARATION

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Signed: 

Martin Hauff

Date: 16-October-2007

ACKNOWLEDGMENTS

> Happy is the person who finds wisdom and gains understanding. For the profit of wisdom is better than silver, and her wages are better than gold. Wisdom is more precious than rubies; nothing you desire can compare with her. She offers you life in her right hand, and riches and honour in her left. She will guide you down delightful paths; all her ways are satisfying. Wisdom is a tree of life to those who embrace her; happy are those who hold her tightly.
>
> Proverbs 3:13-18
>
> (New Living Translation)

I would like to dearly thank my wife and friend Renee Hauff for her support, sacrifice and devotion in encouraging me to pursue this field of study. For several years now she has been forced to *share* me with my research and make sacrifices for which I will be forever thankful. Although it is my name that appears on the cover of this thesis, it would have never been accomplished without her willingness to carry the load whilst I remained missing in action. You are the love of my life and the bride of my youth. I could not have done this without you and all that I am I give to you.

To my supervisor and academic mentor I would like to thank Dr. Paul Beckett. Your counsel and direction have broken me out of my analysis paralysis on numerous occasions and in the times that I could not see a way forward, you have switched on the light to illuminate the path. You have an uncanny knack of being able to navigate the *system* and reduce what would seem like complex processes into a form that even I can understand. I thank you for enticing me back to academia and offering the fertile soil that could produce this thesis.

To RMIT's School of Electrical and Computer Systems Engineering I would also like to offer my sincere thanks for the support you provided by awarding me a scholarship. Having worked professionally for several years prior to returning to RMIT, this research would simply not have been possible had it not been for your financial assistance. Thank you also for being flexible at times when I have needed additional time to complete my submission.

Finally I would like to thank God. It is He who, several years ago, impressed on me the words of Proverbs 3:16 and put the fire in my belly to complete this research. I have no idea where my life will take me but I feel completely confident that it is held tenderly in the loving hands of the creator of the universe.

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

COMPILER DIRECTED CODESIGN FOR FPGA-BASED EMBEDDED SYSTEMS

As embedded systems designers increasingly turn to programmable logic technologies in place of off-the-shelf microprocessors, there is a growing interest in the development of optimised custom processing cores that can be designed on a per-application basis. FPGAs blur the traditional distinction between hardware and software and offer the promise of application specific hardware acceleration. But realizing this in a general sense requires a significant departure from traditional embedded systems development flows. Whereas off-the-shelf processors have a fixed architecture, the same cannot be said of purpose-built FPGA-based processors. With this freedom comes the challenge of empirically determining the optimal boundary point between hardware and software. The fluidity of the hardware/software partition also poses an interesting challenge for compiler developers.

This thesis presents a tool and methodology that addresses these codesign challenges in a new way. Described as 'compiler-directed codesign', it makes use of a suitably modified compiler to help direct the development of a custom processor core on a per-application basis.

By exposing the compiler's internal representation of a compiled target program, visibility into those instructions, and hardware resources, that are most sought after by the compiler can be gained. This information is then used to inform further processor development and assist with hardware/software partitioning. At each design iteration, the machine model is updated to reflect the available hardware resources, the compiler is rebuilt, and the target application is compiled once again. By including the compiler 'in-the-loop' of the processor design, developers can accurately quantify the impact on performance caused by the addition or removal of specific hardware resources and converge to a final solution.

Compiler Directed Codesign has advantages over existing codesign methodologies because it offers both a concrete point from which to begin the partitioning process as well as providing quantifiable and rapid feedback of the merits of different partitioning choices. When applied to an Adaptive PCM Encoder/Decoder case study, the Compiler Directed Codesign technique yielded a custom processor core that was between 36% and 73% smaller, consumed between 11% to 19% less memory, and performed up to 10X faster than comparable general-purpose FPGA-based processor cores.

The conclusion of this work is that a suitably modified compiler can serve a valuable role in directing hardware/software partitioning on a per-application basis.

Chapter 1.

INTRODUCTION

Background

An *Embedded System* can be defined as a combination of computer hardware and software elements, and perhaps additional mechanical or other parts, that have been designed together to perform a dedicated function [1]. Their ability to electronically record, store and process information allows them to provide a layer of *control* and *intelligence* that augments a vast array of products and applications. Compared with other computing platforms, embedded systems are usually purpose built for a specific product and have little functional value outside of the product they serve.

Whilst the term *Embedded System* relates specifically to an electronics sub-assembly, the delineation between what constitutes the embedded system and what constitutes the rest of the product can vary. For example, most modern dishwashers and washing machines contain embedded systems that provide electronic control of the washing cycles. This embedded system would typically be contained on one or more printed circuit boards located behind the machine's control panel and are clearly distinguishable from the rest of the product. At the other extreme, however, is a mobile phone. In this case, it could well be argued that the embedded system *is* the mobile phone due to the complete reliance that the product has on its electronics.

In any case, regardless of what proportion of an electronically controlled product is represented by its embedded system, they are united by some common characteristics. Embedded Systems are integral to the products they serve and must therefore be able to operate across all environmental conditions typically experienced by their host. They are often constrained by one or more factors such as power, weight, size and computational performance. Furthermore, given that they are regularly deployed into high volume applications, the need to decrease their cost often warrants additional optimization effort.

The Rise to Prominence of Microprocessors

In 1971, Intel shocked the electronics world when it announced that the 4004, a 4-bit microprocessor, was available as a single component. Until that time, most people thought of a computer as something that filled a room and certainly not something that would fit inside a single IC. The original 4004 was intended for use in calculators [2, 3] but given its compact form and price, designers quickly found uses for it in other applications.

The 4004 was succeeded in the following year by the 8-bit 8008 but it was the 8080 (released in April 1974) and Motorola's 6800 (released in 1975) that really cemented the microprocessor's dominance [4]. Embedded systems designers embraced the technology and with each new generation of processor came greater integration and more powerful instructions. Microprocessor based systems could be programmed and reprogrammed with far greater flexibility than discrete hardware systems and designers looked more and more to software to implement their system functionality [5].

The embedded processor market flourished throughout the 1970s and by the end of the decade, a full cohort of 4 to 32-bit processors had been released to the market [6]. Since that time, the use of microprocessors in embedded systems has grown at a prolific rate to the point where the 2007 microcontrollers market reached a value of $US13.9 billion [7].

The Role of Compilers in Embedded Systems Development

No one can deny that microprocessors are a technological marvel. But it should also be noted that their success has been heavily supported by equally significant advancements in the software tools and technologies used to program them.

Microprocessors in themselves do nothing but execute a defined set of instructions. The real intelligence of an embedded system stems from the application that runs within it and the increasing complexity found within modern embedded applications is made possible through the use of high-level languages and modern programming techniques [8]. They allow developers to abstract their programs away from the processor's native instruction set and to develop them in a language much closer to our own.

The task of translating a program, written in a high-level language, into something that can be executed by a microprocessor is the responsibility of a *compiler*. Compilers are, in themselves, advanced pieces of software that interpret the high-level meaning of an application and convert it to instructions and operations that can be executed natively on a target system. They perform a range of code optimisations and transformations designed to extract the best

performance out of the target processor whilst shielding the programmer from needing to know anything about the machine they are targeting [9].

There are several benefits to using high-level languages (and therefore compilers) over assembly programming. Because high-level languages are not dependant on the instruction set of a target processor, code written in this way is far more portable and can be reused across multiple projects and designs. Furthermore, because one line of high-level code will often be compiled into several lines of assembly code, coding efficiency is also improved by using high-level languages.

Soft Hardware – A New Computing Paradigm

When the first Field Programmable Gate Arrays (FPGAs) were introduced in 1985, they ushered in a new wave of system programmability. Programmable Logic, of which FPGAs are a part, allow portions of the hardware to be defined in a *soft* manner. Whereas the microprocessor paradigm assumed a fixed hardware architecture that executed a programmable piece of software, Programmable Logic expanded on this to include the ability to *program* the hardware too.

When FPGAs were first released, they filled a niche market and were sold at a premium price. When they found their way into embedded systems it was usually to perform a specific hardware function or to serve as a co-processor that augmented the functions of the main microprocessor. Over time, however, both the manufacturing costs and logic densities achievable in FPGAs have improved to the point where they are now encroaching into the mainstream of embedded systems design. The two major FPGA vendors, Xilinx and Altera [10], have both released downloadable 32-bit processor cores [11, 12] which can be run on FPGA devices that cost less than a dedicated 32-bit processor. There are still some limitations in terms of power, memory density and analogue peripheral capabilities that FPGAs are yet to fully address however device vendors are showing every indication that they will solve these problems within the next 2-5 years. Even now, FPGAs are being used as the main processing platform in some embedded systems and displacing microprocessors from their traditional markets.

The Impact of FPGAs on Embedded Systems Design Flows

To date, design flows for embedded systems have been driven by a need to establish the hardware platform early. There are two primary reasons for this:

1. Hardware development is a time consuming design activity that is also subject to procurement and manufacturing delays.
2. Software testing and some portions of development can only begin in earnest once a hardware platform is available.

In order to minimize the time to market and to maximise the availability of a hardware platform to the software developers, a typical embedded systems design flow will therefore follow these three sequential steps:

1. A requirements analysis would be made to determine the system's functional requirements along with any performance specifications.
2. An off the shelf (OTS) processor that can be used to form the basis of the hardware platform would be identified along with an appropriate compiler if available and the hardware design would begin.
3. System functionality would be implemented in software and compiled to execute on the processor platform.

The dependency on hardware that this design flow is based on makes sense when using off the shelf microprocessors but needs to be reassessed when programmable logic is used. Rather than relying on software to accommodate to the hardware, as is the case in microprocessor based systems, in programmable logic systems the question is now whether the opposite might be possible.

Looking forward to the broader implications of programmable logic technologies, parallels can be drawn from the history of microprocessors. Just as the single-chip CPU led to a shift in the way that embedded systems were designed and implemented, programmable logic devices, such as FPGAs, represent an equally disruptive technology that now challenges microprocessor based design flows [13, 14].

The Tools Required to Capitalize on *Soft* Hardware

In establishing the framework for this research program, I considered a future in which off the shelf processors were completely displaced by programmable logic devices. In particular I was interested in theorizing about the impact that this would have on present day design practices and how they might need to change in order to capitalize on the strengths of programmable logic.

In pursuing this line of questioning, I was also intrigued by a remark attributed to C. A. R. Hoare that suggests that premature optimisation is the root of all evil [15] and that premature local optimisation actually hinders global optimisation [15]. The essence of this wisdom formed the basis of Raymond's Rule of Optimisation: Prototype before polishing. Get it working before you optimise it [15].

The design of an embedded system regularly requires the development (or at least specification) of a hardware platform that executes a software application. It is my assertion that the primary focus of software development is to establish functional correctness whereas the choice of hardware platform is an optimisation choice that attempts to balance cost and performance. If Raymond's Rule of Optimisation were to apply then I contend that software development should precede that of hardware. This is in contrast to the current practice of favouring hardware development during the early portions of embedded systems design flows.

It is therefore the broad goal of this research to devise a means by which software development could lead hardware development. Of particular focus is the role that a compiler will have in such a system and how this compiler can influence, and possibly direct, the construction of the hardware platform using programmable logic.

Summary of Contributions

As a form of hardware/software codesign, Compiler Directed Codesign is a unique approach to the challenge of developing both a custom application along with its hardware platform. By examining the compiler's internal representation of a target program, it is possible to determine those operations that are most sought of by the compiler. Processor development can then focus around supporting those operations rather than trying to offer more universal operations that are of little benefit to the target program.

The specific contributions made by this thesis are as follows:

1. The discovery that compilers can be a useful tool in directing the development of a processor core that is designed to run a specific application.

The traditional role of a compiler is to translate a high-level description of an application into the instruction set of a *known* target processor. This thesis demonstrates that an appropriately modified compiler can also be used to assist with the design of a *new* processor. Compilers have at their disposal complete information about the application being compiled along with

a description of the target processor. By utilizing information extracted from the compiler during the compilation process, the hardware/software partitioning process can be better informed.

2. That the use of a compiler in the codesign process allows dynamic performance metrics of each of the hardware/software solutions to be accurately predicted and compared.

A key feature of the Compiler Directed Codesign methodology is the compilation of the application code into its machine code format for each hardware/software iteration. This machine code can then be used, in conjunction with profiling information taken from a *real* processor, to provide accurate performance predictions for each hardware/software solution. The relative merits of each hardware/software solution can then be objectively analysed.

3. That the Compiler Directed Codesign methodology can be utilized to develop a processor core that consumes less FPGA resources and performs better than general purpose FPGA processor cores.

Because the Compiler Directed Codesign process makes use of a compiled view of the application as part of the codesign process, it only includes instructions in the target processor that are actually used by the compiler. This allows designers to build high-performance processor cores that are specifically designed to support individual applications and without the baggage of supporting unnecessary instructions. The reduced processor complexity leads to a reduced FPGA resource footprint but without adversely impacting the application's performance.

Thesis Organization

The remainder of this thesis has been structured in the following manner:

- Chapter 2 offers a background discussion of the broad field of hardware/software codesign and establishes a framework that positions this work within the current body of knowledge.
- After expanding on the specific challenges facing codesign research, Chapter 3 outlines the supporting arguments for using compilers in codesign activities.
- Chapter 4 lays the framework for a Compiler Directed Codesign platform and identifies the various attributes that a compiler must posses for it to be used in this manner. Reasoning is then provided as to why the GNU Compiler Collection

(GCC) presents as a suitable base upon which the Compiler Directed Codesign apparatus can be built. The later portion of this chapter also discusses some of GCC's limitations along with suitable remedies to minimize their impact on codesign.

- Chapter 5 provides a detailed discussion of the various components developed to support the Compiler Directed Codesign apparatus along with an explanation as to how the compiler's internal representation can be exposed and utilized in the codesign process. The novel use of profiling information as part of the Compiler Directed Codesign methodology is also discussed.
- In Chapter 6, the Compiler Directed Codesign methodology is applied to the custom development of a processor suitable for running an Adaptive PCM encoding/decoding application. A complete decomposition of the iterative codesign approach as presented along with the various measurements recorded as part of the case study. Having identified a suitable quality factor, a instruction set solution is proposed and then used as the basis for building a custom processor core. This core is then synthesized for an FPGA and compared against other commercially available cores.
- Chapter 7 concludes the thesis with its findings. A discussion of the merits and demerits of Compiler Directed Codesign is presented along with opportunities for further work.

Chapter 2.

BACKGROUND AND RELATED WORK

Before embarking on a discussion of the relative merits of various approaches to the codesign challenge, this chapter begins by providing a suitable definition of codesign. Existing approaches to the codesign problem will be presented from the literature and how they have influenced the approach taken in this thesis will be discussed. The chapter will conclude with a closer examination of the specific issues that have motivated this course of research.

What is Codesign?

The term hardware/software codesign surfaced in the early 1990s to describe a confluence of problems in integrated circuit (IC) design [16]. The prefix *co* is used to denote togetherness or a joint relationship and its use within the term *codesign* signifies a joint or partnered development of both hardware and software. A classic interpretation suggests a true interaction between the hardware and software design paths as opposed to the coincidental development of hardware and software at the same time. It encompasses the broader fields of ASIP synthesis (Application Specific Instruction-set Processor), execution acceleration by means of custom-computing machines, design of System-on-a-chip (ASICs with embedded processor cores), embedded-systems design [17], and hardware/software co-verification processes [18].

In order to clarify how the codesign term will apply to this thesis, I will make use of another word – *conjoint*. This term more fully describes the notion of two entities being united with one another in such a way that they are inseparable and without clear delineation. By this definition, a *conjoint system* is one comprising both hardware and software elements that combine inextricably to form a complete product. As such, it refers to a state of *being* as opposed to the process by which it was created. On the basis of this, the term *codesign* denotes the process by which a conjoint system is created.

At this point, it is worth dwelling on the subtle distinction that is being made here. Whereas the literature would support the usage of *codesign* to describe design processes that separate

hardware and software so that they can be developed in parallel (but in isolation) from one another, it is my assertion that this approach is at odds with the definition of a conjoint system – i.e. that the hardware and software are inextricably connected. If the notion of a conjoint hardware/software system is to be truly embraced, then the development process must seek to develop the two in lock step with one another.

This presents the interesting challenge of how to seamlessly bring together hardware and software throughout a product's development. Regardless of whether the focus of development is around ASIPs, execution acceleration through instruction extensions, custom hardware blocks, or any other endeavour that requires hardware and software development, the challenge remains.

Compilers provide the mechanism to map software functionality onto the available hardware platform and thereby provide the glue between the two. This thesis is titled 'Compiler-Directed Codesign' because it takes the position that the function fulfilled by compilers offers the means by which hardware and software can be truly co-designed. Because they sit at the junction between hardware and software, it is the assertion of this thesis that they can also provide a central point of analysis that is useful for directing codesign development. The body of this thesis will seek to demonstrate how a suitably modified compiler can be used in this fashion.

Traditional Codesign Flow

Based on adaptations taken from [19] and [20], the typical development flow for a Hardware/Software system is described below:

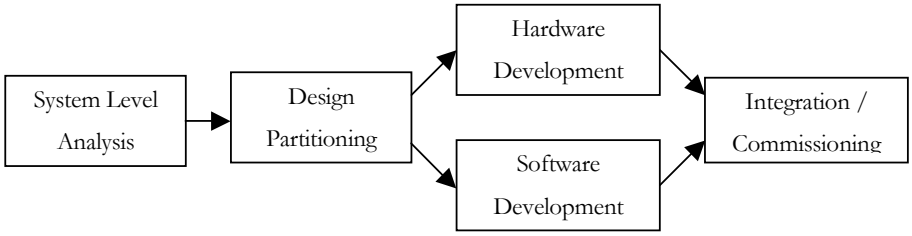


Figure 1 Traditional codesign development flow

As shown in Figure 1, the major blocks that make up the traditional development flow are:

1. *System level analysis:* The functional requirements of the system are identified and performance parameters determined.
2. *Design Partitioning:* The product is subdivided into hardware and software components that can be developed in parallel.

3. *Parallel Hardware/Software Development:* In order to minimize time-to-market, it is advantageous to make use of parallel development streams. During this phase both hardware and software would be implemented by (typically) two separate development teams.
4. *Integration/Commissioning:* The Hardware and Software components are once again combined together to form the completed conjoint system. If the preceding design stages have been adequate then the conjoint system will be deemed a success. If not then lengthy analysis and rework efforts may ensue.

Although all phases in the design process are critical to the product's success, the focus of this work is specifically in the Design Partitioning phase. The interest lies in the fact that this stage represents the difficult task of finding a division between two elements which, by definition of a conjoint system, are inextricable. In order for the design partitioning phase to be successful, it must seek to manage the interrelationships that occur between hardware and software and it must do so in a manner that satisfies the system level constraints.
To make matters worse, any error introduced at this phase will often not reveal itself until the integration phase. In larger projects, this phase may not occur until several man-years of effort have been expended.

Design Partitioning

Design partitioning is pivotal to the success of codesign and has been the subject of many hundreds of research papers over the past decade. A summary of several system partitioning procedures can be found in [17]. In essence, the primary goal of any partitioning process is to determine and segment those portions of the product's functionality that will be implemented as software routines and those that will be implemented in hardware. Much weight needs to be given to the partitioning decision as it guides the succeeding software and hardware development efforts. Poorly partitioned systems will subsequently lead to suboptimal performance and/or increased product cost.

Structural Versus Functional Partitioning

Design partitioning can be done on the basis of either structure or function as indicated in Figure 2.

- **Function** concerns itself with *what* a block does.
- **Structure** deals with *how* that block is implementd.

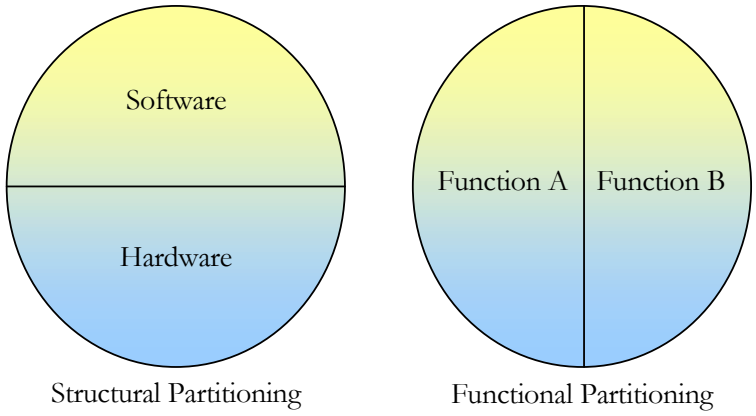


Figure 2 Structural versus Functional Partitioning

In the realm of hardware/software codesign, *function* defines the task(s) that the system is intended to perform and *structure* defines what mixture of hardware and software will be employed to perform those tasks.

There is evidence to suggest that functional partitioning is the better approach for hardware/software codesign [21]. This is because it allows for a more complete exploration of the design space since functional blocks can have both a software solution as well as a hardware implementation.

Given that the intended outcome of the partitioning process is, in fact, a structural definition of the system, one might wonder how this is possible when partitioning is made based on functionality. This can be accounted for by simply stating that the partitioning process is an iterative one. Although it begins by first dividing the design according to function, it concludes by determining which of those functions are to be deployed as hardware or software.

It is important to keep this in mind at the specification stage of a project as any system level blocks that have been formed with structural assumptions in mind may unduly prejudice the partitioning process and lead to a suboptimal result [15].

Evaluating Partitioning Research

Hardware/software codesign is by no means a recent field of research and there have been many contributions to the current body of knowledge. The field is vast both in terms of the volume of contributions and their breadth of coverage [17]. In order to gain a better grasp of the various approaches to hardware/software codesign and to define where they sit within the current body of knowledge, the next two sections will provide an overview of that work.

The discussion is broadly ordered around aspects of Granularity (the size of the smallest unit that can move between hardware and software) and the Optimisation Goal.

Evaluating Partitioning Research – Granularity

In the context of hardware/software codesign, *granularity* defines the minimum *size* that candidates will be considered for migration between hardware and software. Moving from largest to smallest, typical granularity sizes are:

- Tasks (coarse grained)
- Functional blocks
- Loops
- Basic block (fine grained)

Task[1] Level Granularity

In embedded systems, a *task* refers to a self-contained or concurrent thread of execution that operates with a high degree of independence from other execution threads within the system. Where cross-communication is required with other tasks it is usually implemented through well defined interfaces. Because of these neat boundaries, tasks are often considered as attractive candidates when performing hardware/software partitioning as they can be readily isolated from one another.

Given a system with multiple tasks, the focus of task level hardware/software codesign is to identify which tasks would best be placed in hardware and which tasks would best execute as software. Tasks do not have the option of straddling between hardware and software unless they are further decomposed into sub-tasks.

The primary weakness of this approach is that the coarser grain size can lead to less exacting partitioning placement but it has been argued that this can be controlled, to some extent, by controlling the size of the individual tasks [22]. Some success has been reported in this area using dynamically reconfigurable hardware to implement the individual tasks [23] whilst others have considered the impact on inter-task communications for Task Level Partitioned systems [24] as well as for tasks running on different processors [25].

[1] Whilst the term *task* is commonly used in embedded systems literature, the term *thread* is commonly found in computer science literature. In general, a *task* relates to an independent execution unit typically found in hard real time systems such as control applications. A *thread* relates to an independent execution unit found in data flow applications where the same hard real time constraints do not exist.

One complication that is regularly encountered when attempting Task Level Partitioning occurs when common functions are used by multiple tasks as they must be able to preserve separate instances of their local data. In a software-only environment this is achieved by making use of different data spaces on the system stack but when functions are migrated into hardware, localised data is commonly stored in registers that are interspersed throughout the hardware implementation and more difficult to disentangle into something as convenient as a memory stack.

In order to guarantee re-entrancy when one task is migrated to hardware and another remains in software, common functions can be instantiated in both software and hardware however the optimality of such a solution would be questionable. Alternatively common functions could be described as their own task in which case they would become available to both hardware and software routines and could block if two tasks attempted to access them simultaneously [22]. However such an approach would represent a significant departure from the basic concept of task level partitioning and encroaches into the next granularity level which is function level partitioning.

Function Level Granularity

Function Level Partitioning recognizes the limitations of Task Level Partitioning and attempts to find a better utilisation of the available hardware resource. By allowing each function to be implemented in either hardware or software, common functions can be deployed in hardware and used by different tasks if necessary [21, 26-28].

One problem that begins to emerge as the partitioning granularity moves towards finer granularity is that the various issues of scheduling become more complex. Functions ported into hardware are generally not designed to be re-entrant which means that the processes calling the hardware functions are also not re-entrant. This complicates the task of real time operating systems as they no longer have the flexibility to arbitrarily swap tasks in and out of execution if it is possible that two tasks might wish to call the same hardware function.

The use of mutual exclusion primitives and process blocking is an effective means of preventing multiple tasks from simultaneously accessing a shared resource however this needs to be monitored closely to ensure that it does not become the focus of performance bottlenecks. The performance benefits of moving functions to hardware will quickly be negated if they are blocked by other functions accessing common resources.

Loop Level Granularity

Loops often represent a significant portion of application execution time whilst often being compact enough to implement using modest hardware resources [29]. This has been the focus of the PICO-N group from Hewlett Packard [30].

Loop Level Granularity offers some promise as a point of partitioning however its benefits are heavily dependant on the nature of the application. DSP style applications that operate on streams of data are particularly well suited to Loop Level Partitioning since a large proportion of the application involves recurring loops. Control style applications profit less from Loop Level Partitioning because they simply have fewer loops.

A further impediment to deploying Loop Level Partitioning is the complexity of loops. Loop Level Partitioning functions best on well defined loops with minimal conditional branching within them. As branching complexity increases, so too does the hardware complexity required to implement them. If the control path through the loop is reliant on data from slower software routines, the performance benefits of hardware deployment can be all but lost as the loop processes are left waiting to be 'fed'.

Recursive loops are generally not transferable to hardware and therefore do not profit from Loop Level Partitioning.

Basic Block Granularity

Basic Blocks are those portions of an application's execution path that do not contain any conditional branches. Their execution speed is static and easily determinable regardless of the data they are operating on. Most research that refers to fine-grained partitioning is referring to Basic Block Partitioning.

A number of metrics that can be used to analyse the suitability of porting basic blocks written in ANSI C to hardware have been proposed [31]. Basic Block Level Partitioning has been used to dynamically tune the performance and energy consumption of embedded systems [32] and, in another system, basic blocks were extracted and executed as a single instruction by an FPGA acting as a co-processor [33].

Although Basic Block Granularity addresses the conditional branching issues that plague Loop Level Granularity approaches, its success is still heavily dependant on the application and can not be applied in all situations. Basic Block Granularity must still consider issues of re-entrancy, process concurrency, and utilisation rates of basic blocks that are implemented in hardware.

Instruction Level Granularity

Below Basic Blocks, the next partitioning point is at the Instruction Level. An interesting transition occurs in the literature at this granularity level. It would appear that an artificial boundary occurs between research reported under the banner of hardware/software codesign and research reported as Application Specific Instruction Processor (ASIP) development. The point of demarcation appears to be the instruction set however in line with [34], the position I have taken in this thesis is that ASIP development is really a form of ultra-fine grained partitioning and should therefore be included in the discussion of granularity.

ASIP development has been in existence for some time and remains a very active field of study. According to [35], the synthesis of ASIP systems basically consists of three sub-problems: microarchitecture design, instruction set design, and instruction set mapping. Using High Level Synthesis (HLS) techniques, microarchitectures can be developed at the RTL level from behaviour specifications [36-38]. Instruction Set Synthesis (ISS) is used for instruction set design [39-41] and Instruction Set Mapping (ISM) compiles the given applications to assembly code [42].

More recently, ASIP research has moved its focus towards reconfiguring extensible processors. Some notable contributions that will be further discussed under the heading of 'Other Closely Aligned Work' in Chapter 3 are Tensilica's Xtensa [43-45], and the PEAS project [46, 47]. Other examples of extensible processors include Nios [48], the PowerPC [49], ARM [50], and Motorola's S-Core [51]. With the exception of PEAS, all of these contributions work from a predefined base architecture that is extended through the judicious fusing of commonly occurring instruction sequences into single instructions. The common goal is speedup within specific area constraints and they assume that the base processor is unable to achieve its performance goals without instruction extension. This thesis explores the opposite end of that spectrum where minimal area is the goal rather than speed alone. So rather than adding hardware to achieve speedup, this thesis explores the process by which processor hardware can be removed whilst still achieving a minimal level of performance. This is not possible with ASIP design solutions that use a base processor instruction set that cannot be pruned [52].

Additional work relating to the use of programmable logic that is tightly coupled to an extensible VLIW processor is presented in [53]. Through this mechanism, the designer can develop custom hardware in the FPGA and make use of it via extended instructions. The

research focus was on determining the optimum extensions for a given application but it did not go as far as showing how the instruction extensions would be revealed to, and used by, a high-level compiler. Furthermore, given the advances in FPGA capacity, it would seem warranted to place the entire processor core inside the FPGA rather than just using it as a coprocessor. This would enable the removal of the VLIW processor from the circuit.

Atmel have commercially released their FPSLIC processor that combines a fixed processor core with a sea of reconfigurable logic and this has been the focus of some research into runtime reconfiguration opportunities [54]. Xilinx has also released a number of high-end FPGA products that include one or more hardwired PowerPC cores and a study of instruction set extensions for a fixed processor that contains reconfigurable logic has been undertaken [55].

Perhaps we will see more mixed mode devices in the future but presently the tools do not support the use of these devices in an *extendible instruction set* mode. Instead, it would appear that device vendors are offering these devices so that designers can minimise board space whilst still having access to both a traditional processor and reconfigurable logic.

Performance Effects due to the Processor's Register Set

In general, processing platforms that utilize a memory caching system will run faster as the memory page size increases. This is due to the processor being able to keep more data in its cache which in turn leads to fewer page faults. Experiments have, however, revealed cases in which the reverse is true; a decrease in the size of the store is accompanied by a decrease in running time [56]. The results of this anomaly can be equally applied when considering the quantity of registers used within a processor when register spilling occurs as part of application execution. But in spite of this anomaly, research has shown that varying the register set size can have a marked improvement on the performance and power of a processor [57] across a number of benchmark applications.

In light of this, a study was performed in [58] that examined the interaction between a processor's register set size and the compiler's code generation strategy. It was found that the code generation strategy employed by the compiler had a far greater effect on overall performance than did the register set size. The conclusion that can be drawn from this is that architectural modifications are of little use if the compiler's code generation strategy does not (or cannot) make use of them. This highlights the importance of including compiler development as part of an ultra fine grained partitioning approach.

Global versus Local Optimisations

In determining the best granularity to use for hardware/software codesign partitioning, it is worthwhile considering the relationship between global and local optimization efforts. Amdahl [59] has shown that even when the fraction of serial work in a given problem is small, say *s*, the maximum speedup obtainable from even an infinite number of parallel processors is only 1/*s*. If *N* is the number of processors, *s* is the amount of time spent (by a serial processor) on serial parts of a program and *p* is the amount of time spent (by a serial processor) on parts of the program that can be done in parallel, then Amdahl's law says that speedup is given by:

$$Speedup = (s + p) / (s + p / N)$$
$$= 1 / (s + p / N)$$

Equation 1: Amdahl's law

Amdahl's law is important because it provides a relationship between the effect that local optimisations have on the global performance. It implies that the maximum speedup achievable by optimising a given portion of an application is limited by the extent to which that portion is used within the overall application. The inference is that optimisation efforts should focus on those portions of code that are most used by the application.

Evaluating Partitioning Research - Optimisation Goal

It is well established that judiciously adding hardware resources to a given embedded system will generally result in a speedup of the application – i.e., it will allow a given task to be completed faster. However embedded systems are not measured by their speed alone. Other pertinent metrics include power consumption, size, weight, and cost. Embedded developers are required to carefully balance all of these metrics and while it may be true that increased hardware resources will lead to an application speedup, it may also lead to:

- Increased power consumption/heat dissipation
- Increased chip/board size and weight.
- Increased component costs

On the other hand, additional hardware resources may also offer:

- Lower clocking frequencies
- Lower program memory requirements
- Additional functionality

Because of the vast array of embedded systems applications, the inter-relationship of competing requirements will be different for every embedded product and no single rule is sufficient to cover all circumstances. Subsequently researchers have experimented with a number of optimisation strategies.

Cost/Performance (Speed)

The starting point for most optimisation strategies involves trading off cost with performance. In this context, *performance* generally relates to a device's ability to function in a timely fashion and *cost* relates to the piece-price cost of the end unit.

Different research efforts have attacked this optimisation problem from different angles and although they are all heavily interrelated, the differing emphases lead to slightly different solutions; where one research group might try to maximise performance for a given price, other researchers may seek to minimize cost for a given level of performance.

Minimizing Cost

A first order approach to cost minimisation is to assign an application's software tasks to a selected set of hardware components and processors such that the resultant system is optimised in terms of cost and performance, while satisfying all the given constraints [60]. The key point to this analysis is that any assessment of cost or performance must be considered in the context of the system's requirements. Increasing the speed of the system beyond what is required is wasted effort as it will not improve the user's perception of the product nor will it improve the product's quality or function. The goal, therefore, is to minimize cost whilst *still achieving a minimum standard of performance.*

To accomplish this goal, the approach taken by D'Ambrosio and Hu in [60] was to break a candidate design into discrete self-contained functional units and, where possible, produce both hardware and software implementations for each. By costing the hardware resources required for each implementation and considering the speed that each implementation offered, a cost/performance optimisation equation was devised that offers a pareto-optimal set of solutions from which a systems designer can choose a final solution.

One limitation that the authors acknowledged in this method is the assumption that communications between functional units involves zero delay. A second limitation is the prohibitive complexity that ensues if the designer wishes to increase the number of attributes to optimise the design across. Considering that this work was conducted over 10 years ago, it

is possible that current computing capabilities may have overcome (or at least reduced) the impact of this latter limitation.

In a more recent study, a costing model was developed that considered costs beyond that of unit cost alone. In [61], a detailed costing model was proposed that included various hardware foundry combinations, different production quantities and different levels of design reuse. This approach sought to take a more encompassing approach by evaluating system (both hardware and software) development, fabrication, and testing costs concurrent with hardware/software partitioning in a codesign environment.
The work was reliant on estimation models and also requires the system designer to determine the specific system requirements so that accurate attribute data can be predicted. Subsequently the approach is best suited to well defined systems.

Because the cost of a processing platform is proportional to its area, one indirect cost minimization scheme involves reducing the size of the processor. In [62], researchers sought to migrate a CISC processor to an ASIP architecture. As part of this process, they performed an analysis of existing programs and found that around 50 of the processor's instructions were never used which, if removed, would yield an area saving of around 40%. The argument for reducing instructions on a processor is by no means a new one as it was the premise of the introduction of RISC processors. But the use of this instruction set reduction technique to re-design an old processor to make it more suitable for system on chip applications was still considered worthy of further exploration.

Maximising Performance

Focussing on cost minimisation is only a worthwhile pursuit once it can be guaranteed that minimum standards of performance can be achieved. For some advanced systems, however, this cannot be assumed. Some embedded systems are more heavily constrained by hardware performance than by cost. In these cases, the designer may seek to maximise system performance for a given price.
An early example of this approach identified critical regions of software in a given application and ported them to FPGA hardware to improve system performance [63]. In this instance the specific hardware components, namely the microprocessor and FPGA, were locked into the design and it was considered the system engineer's task to extract the maximum

performance with the given architecture. Cost did not change – only the available performance.

This approach to codesign has become more feasible as configurable logic devices such as FPGAs have increased in capacity and decreased in cost. Programmable logic can offer an important contingency option to insure against an under performing processor by allowing some of the processor's computing burden to be off loaded into custom hardware. As a brute force approach it is quite valid and possibly very successful where rapid time to market is a key constraint. However, its usefulness is restricted to coarser grained partitioning as will be discussed in a later section.

Power/Performance

Mobile embedded systems often have tight power budgets and must be capable of operating for extended periods of time from a battery source. Excessive power consumption will require larger and heavier batteries that increase the cost and compromise the portability of the device. Rather than optimising for cost and performance, in this situation the objective is to optimise for power and performance.

One approach to this problem [64] is to attempt to minimize quiescent current by maximising the average utilisation rate of various hardware resources. That work assumes that processor resources that are powered up, but left idle, consume power with little or no benefit to the application. By adjusting the instruction set to maximise usage (or conversely to minimize idle time) of key, power-consuming resources, power reductions in the range of 35% to 94% were reported in [64].

An alternate approach is to use clock-gating and frequency scaling to throttle or switch off unused portions of the hardware (e.g., [65]). For this to be successful, the hardware needs to be partitioned into clock domains that can be readily activated and deactivated and so whilst it considers hardware partitioning to some extent, it does not specifically address the issue of hardware/software partitioning.

The optimisation of designs according to power and performance is a very important topic in itself and one that has attracted much research that extends well beyond hardware/software codesign. It is not, however, the focus of this thesis and will not be considered in any further detail.

Design Reuse

Design reuse makes good logical sense as it seeks to use existing design components as a leveraging point for creating additional designs. If a new design can be constructed from a number of pre-existing and pre-verified components then the risk borne by the new design is lowered and development time can be reduced. The economic argument for design reuse is based on reducing non-recurring engineering expenses as opposed to minimizing the recurring engineering and parts cost of the completed system.

A complete codesign environment for embedded systems which combines automatic partitioning with reuse from a module database has been described in [66]. In this environment, new designs are constructed from a selection of pre-developed hardware and software modules that have already been fully characterized and qualified.

Although the authors reported that their hardware/software solution performed better than a software only solution, they did not offer any evidence of the optimality of their design when compared with a design that does not specifically attempt to make use of design reuse.

In many respects, design based on reuse can be viewed as a library based design methodology in that a new design is constructed from a *library* of existing modules. This approach to hardware/software codesign will be discussed in more detail in a later section along with other partitioning methodologies.

Memory-CPU optimisation

The notion of optimising memory against the CPU size is an interesting version of a cost/performance optimisation strategy. The idea is based around exploring memory cost reductions that can be achieved when using a CPU core that is smaller in precision than the largest variable in the application [67]. Since the decreased CPU precision can lead to a smaller memory footprint for the processor's instructions, cost reductions of as much as 30% were reported in [67] when compared with the single-precision point cost.

Estimation Techniques

In order to pursue a specific optimisation goal, the designer must have access to metrics that can be used to indicate the impact of various competing partitioning decisions. In a system that is still under construction, it may be neither feasible nor possible to obtain these metrics through direct measurements and so designers may use estimation techniques instead. In the same manner that there are many goals against which a system may be optimised, there are equally as many parameters that require estimation models.

Absolute accuracy is not always the goal of an estimation technique. In some situations, the results of the estimation process only serve to guide further development and so relative accuracy may be sufficient. In fact, where estimation techniques are used as part of design exploration, the speed of the estimation technique is just as important as the relative accuracy of the results [68].

A brief discussion of several estimation techniques and the metrics they seek to forecast follows:

Estimating Execution Speed

A first order analysis of execution speed is presented in [60] by dividing the number of executed instructions by the MIPS rating of the target system. This model is extended in [69] where the authors consider the number of execution cycles needed for each instruction in the program, the number of memory read/writes, and the number of cycles per memory access. More recently in [70], statistical terms have been combined with deterministic information to predict execution times for complex CPUs to within 3%-4% accuracy.

In addition to speed estimates, [71] included estimates of the implementation cost of the hardware portion of a mixed hardware/software system. It combined information from data dependency and timing graphs to produce a mathematical model that was capable of considering task parallelism and hardware sharing.

Estimating Area

When considering the viability of moving functionality into hardware, it is helpful to know how much silicon area will be consumed by that functionality. This information can then be used in one of two ways. For systems that assume a fixed processor connected to a dedicated reconfigurable FPGA, the cost of the configurable computing resource is already paid for and so the focus is on maximising its use. In order to do this, as much functionality as possible should be pushed into the FPGA and executed as hardware rather than software. Put another way, the performance needs to be maximized for a given area constraint. This problem was tackled in part by [72]. In that work, the authors used area estimates of the various functions that were eligible for hardware execution and sought to fit them within the available configurable resources using a modified Knapsack algorithm [73].

The second way that area estimates are useful is in absolute terms where the total silicon area of a given core needs to be minimized whilst still meeting performance requirements. [74] used this approach in their work with the Tensilica processor as did [75] in their work with

high-level estimation techniques and flexible granularity sizes. This approach is best suited to instances where the hardware platform is not fixed and each additional unit of silicon area adds to the overall cost of the final product.

Estimating Power

In estimating the power consumed by an embedded system, [76] includes the contributions made by both the hardware and software components. The hardware model attempts to relate the average power dissipation of the VHDL descriptions to the physical capacitance and the switching activity of the design nets. The software model makes use of a compiled intermediate form of the target application and applies a power metric to each instruction. Using an intermediate instruction representation has the benefit of processor independence but can also be a source of inaccuracies that are dependent on how the intermediate instructions are mapped to the target system.

As processors have increased in complexity, power estimates have needed to become more sophisticated to account for contributions made from multiple memories and caching systems. In [77] the authors incorporated power metrics into an instruction set simulator. As the target application was simulated, an accurate power figure could be determined based on information taken from the CPU core, RAM/ROM, cache memory and application-specific hardware.

Profiling Techniques

The complexity of many conjoint systems is such that it is not always feasible to attempt to model or estimate which portions of the application should be implemented as hardware or as software. In terms of hardware/software codesign, profiling is one technique that can be used to identify computationally expensive kernels that would yield good performance improvements if they were ported into hardware [78].

Profiling is the process of analysing a running application and building a profile of which software routines consume the greatest proportion of processing time. This information can then be used to determine which units would most likely yield the best improvements given additional attention and/or resources.

Traditionally, profiling is done using dynamic execution information extracted from a running application. This yields the best results when combined with a code coverage tool as it allows the designer to see the actual execution paths taken through software branching points and locate inefficient routines.

Of course, in order to dynamically profile an application one must first have access to an application platform (i.e. hardware) or at least a suitable simulation model for it. But given that the one of the objectives of hardware/software codesign is to determine the hardware, there remains the considerable challenge of profiling an application on hardware that doesn't yet exist. To address this issue, some recent research attempted to characterize the flow of control of the application under static conditions [79]. This technique was aimed at supporting early performance evaluation of embedded software for which traditional profiling tools are not practical.

One of the benefits of applying a profiling approach to performance management is that it allows the designer to focus first on getting the system functionality correct and relegates performance to a secondary goal. Not only does this approach adhere to Raymond's Rule of Optimisation [15] that was quoted in Chapter 1, but it also ensures that a complete understanding of the system's requirements is fully understood before optimisation decisions are made.

Modelling Conjoint Systems

When designing complex systems, the use of abstraction techniques that reduce the key elements of the design into more manageable portions is often helpful. These design units allow the designer to model information that is pertinent to the chosen optimisation goal while filtering out any unnecessary detail. At the highest level of abstraction, models can be useful as a design exploration tool under which various scenarios can be tested. As the design develops, the detail of some models can be developed to the point where a complete hardware/software solution can be realised.

Modelling Frameworks

At the very highest level of abstraction are modelling frameworks. They define metamodels that represent the relationships between different models and modelling languages. The Kernel Language [80], RPM Metamodel [81] and Ptolemy project [82] are all examples of modelling frameworks that have been designed to support heterogenous modelling and design based on multiple Models of Computation (MOCs). The Sonora environment [83] is yet another modelling framework designed primarily to implement their own model-based codesign methodology described in [84].

System-Level Modelling Languages

At the next lower level of abstraction are the modelling languages themselves. MOOSE [85] is a Model-based Object Oriented Systems Engineering method. It draws from the well established software principles of encapsulation and abstraction found in traditional object-oriented software development and applies them to the codesign of embedded systems.

MCSE [86] is a modelling system that attempts to capture all of the functional and non-functional aspects of the system and allow the designer to progressively refine the model throughout the development process until a prototype hardware/software solution can be generated from the system definition.

TTL or T-LOTOS (Templated LOTOS) language is an extension of the LOTOS language to make it suitable for hardware/software codesign [87, 88]. The basic idea is that the behaviour of the system can be described by observing from the outside the temporal order in which events occur.

SDL is a Specification and Description Language that has been used by a number of researchers [89-91] in codesign activities. SDL is an object-oriented language standardized by the International Telecommunication Union Standardization sector (ITU-T) and is designed for specifying reactive systems [92]. The approach of using SDL for codesign activities has the benefit of being able to synthesize VHDL directly [93] however according to [94] the resulting VHDL is suboptimal. This lead to the conclusion that the hardware parts of computational systems should be described with HDLs leaving SDL to describe software and communication between asynchronous systems.

ESTEREL [95] is another language dedicated to programming reactive systems and can be used to create finite state machines (FSM). It can be used as an input to the POLIS design environment [96] and has been used by several researchers in this manner for codesign activities [33, 97, 98].

Using Existing Hardware or Software Languages for System-Level Modelling

Given that highly capable hardware and software languages exist and are already being used successfully in the development of the hardware and software components of conjoint systems, it would seem desirable to find a way to augment these languages to make them suitable for system-level activities.

COMET [27] is a function level codesign system that accepts C and VHDL functions along with some rule files to create the system specification. The focus of this work, however,

appears more on co-*simulation* rather than traditional codesign as it does not appear capable of transferring functionality between hardware and software.

VIOOL [99] is a tool suite that is able to translate VHDL code into C++ classes that can then be instantiated in a main routine to form a hardware simulator. Like COMET though, the authors refer to it as a codesign tool but its focus remains on co-*simulation*.

A more promising approach is given in [100] where an extended form of VHDL (X'VHDL) is used for the input specification. This is then transformed into an extended timed Petri-Net notation based model that is capable of representing control flow in both the hardware and software domains. After a process of optimisations and transformations, the system is then able to generate a complete solution that includes the RTL hardware implementation and C program.

The use of system-level modelling in the development of conjoint systems is a worthy research endeavour as it allows for more global optimisation opportunities to be explored. However regardless of which modelling system is used, it must be capable of producing synthesizable hardware and software modules that can be used to implement the final system. At that level, a compiler will be required to map the software executable onto the hardware platform.

Summary

This chapter has provided a broad overview of the field of hardware/software codesign. For the purposes of this thesis, hardware/software codesign shall relate to joint development and optimisation of a system comprising of both hardware and software components. While the overall goal of the partitioning process is to define the structural boundary between hardware and software, it is based on the functional blocks within the system.

Several approaches to partitioning have been attempted and can be broadly grouped according to their granularity and the optimisation goal. Granularity relates to the size of the minimum element that can be moved between hardware and software and ranges from course grained, task level partitioning through to fine grained, basic block partitioning. Ultra-fine grained partitioning at the instruction level is a natural extension of this continuum.

The optimisation goal seeks to find the *best* solution that balances constraints such as size, power, cost and performance. To help this process, direct measurements taken from similar systems or estimation techniques can be used to predict the impact that design tradeoffs will have.

The greatest benefit is derived when optimisations are made at a global rather than local level. This requires a unified system-level definition that encompasses both the hardware and software portions of the design. To this end, a number of modelling approaches have been proposed that range from highly abstracted modelling frameworks through to the existing hardware and software languages. However regardless of how the designer chooses to represent their design at a system level, a compiler will invariably be required at the final stages of implementation so that the resulting software can be mapped onto the hardware platform.

On the basis of all of this, the focus of this thesis will be as follows:

- Hardware/software partitioning should be based first and foremost around function rather than structure [21].
- The optimisation goal is dependent on the final application and the choice of which strategy to pursue for further research is somewhat arbitrary. Given this freedom, I have chosen to draw from my background in automotive electronics and focus on minimizing the costs of high-volume embedded systems. This goal is similar to that found in [60] but I intend to take a very different approach to the functional decomposition method presented in that work.
- Whilst modelling and estimation techniques provide excellent starting points for hardware/software codesign, a profiling methodology draws from actual run time data and appears to be a pragmatic approach that is already well proven [78]. In making this choice, I acknowledge the difficulties associated with using a profiling approach in systems where the final execution platform is still the subject of development. One of the focal points of this thesis will therefore be to determine how this limitation can be overcome.
- The implication of Amdahl's law [59] is that optimization efforts should focus on those portions of the application that are most heavily used. The partitioning approaches presented in [23, 27, 33] are all based around using custom hardware to offload some of the processing burden from the main processor. Although they target computationally expensive portions of the overall application, they are essentially a localized form of optimisation. By shifting the focus to the processor's instruction set allows for the possibility of making enhancements that are of benefit to the entire application and not just portions of it. This is exhibited in [43-45, 48-51, 53-55], however they all focus on *extending* an existing processing platform and do not

offer the option of pruning the base architecture [52]. This thesis will move beyond that work by building the processing platform's instruction set from the ground up and thereby implementing a form of ultra-fine grained hardware/software partitioning.

- The work of [58] highlights the significance that the compiler has in the performance of the overall application. Given that it is the compiler's responsibility to efficiently map software functionality onto the available hardware resources, it must be included as integral to hardware/software codesign. It is located at the pivot point between hardware and software and promises the best means to measure the impact that various instruction choices will have on performance.

Chapter 3.

EXPANDING ON THE CODESIGN CHALLENGE

In the previous two chapters the broad objectives of this research were laid out in the context of existing work. This chapter will further expand on the specific nature of the codesign problem and provide some additional detail as to how one might approach a solution. The role of compilers in traditional software systems will also be discussed along with how their attributes make them well suited for general use in codesign activities.

Managing Hardware/Software Tradeoffs

As the field of software has evolved and matured from its early beginnings in the late 1950s [5], the way that it has been represented has changed significantly however its role in computing systems has remained unchanged. Software relates to the set of instructions or *program* that is applied to a fixed computing platform so that it can be manipulated to perform a specific task. It is differentiated from hardware in the sense that hardware is rigid and not easily changed whereas software can be readily updated to bring new and higher levels of functionality to a hardware platform.

Whereas early software languages and systems were tightly coupled to the underlying hardware they were controlling, modern software is written using high-level languages and is abstracted far from the processing platform. An examination of software in this form reveals a stark contrast with hardware.

Hardware is a concurrent medium that is capable of performing a number of functions simultaneously. Functionality implemented in hardware alone can capitalize on the concurrent nature of the hardware to rapidly complete a given task. Software, on the other hand, is essentially sequential and each instruction in the software program must share access to the underlying hardware resource. The trade-off between these two domains is that whilst hardware can execute a function rapidly, it is far more costly to develop and each parallel path through the hardware consumes additional silicon resources. As an alternative, what software lacks in speed it makes up for in cost reduction. The key problem of

hardware/software codesign revolves around determining the appropriate hardware/software mix that will yield the desired performance whilst minimizing cost.

A simple demonstration of how the benefits of hardware and software can be traded off against one another can be shown through a simple 32-bit addition function:

$$A_{31:0} + B_{31:0} => C_{31:0} + C_{out}$$

A hardware only implementation of this function would use a single, 32-bit combinatorial adder component (see Figure 3).

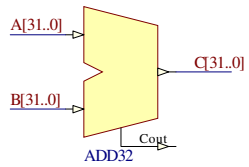


Figure 3 32-bit adder

The commutative nature of addition reveals that the same functionality could equally be performed as two successive 16-bit additions:

$$A_{15:0} + B_{15:0} => C_{15:0} + C_{16'}$$

$$A_{31:16} + B_{31:16} + C_{16'} => C_{31:0} + C_{out}$$

In this case, only a 16-bit combinatorial adder component is required but it is *shared* between the high and low word additions (see Figure 4). Also the Carry ($C_{16'}$) generated during the low-word addition must be propagated to the high-word addition. Performing two, 16-bit additions will take approximately double the time of a single, 32-bit addition.

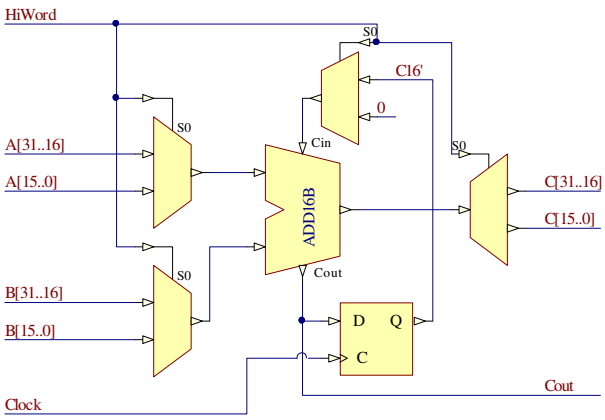


Figure 4 Performing 32-bit addition using a single, 16-bit adder

Further reduction of the above equation could be made to only utilize an 8-bit combinatorial adder component as follows:

$A_{7:0} + B_{7:0} => C_{7:0} + C_{8'}$

$A_{15:8} + B_{15:8} + C_{8'} => C_{15:8} + C_{16'}$

$A_{23:16} + B_{23:16} + C_{16'} => C_{23:16} + C_{24'}$

$A_{31:24} + B_{31:24} + C_{24'} => C_{31:24} + C_{out}$

Once again, a smaller hardware adder can be used but the time to execute the 32-bit addition using only 8-bit hardware will take approximately four times that of the single 32-bit addition.

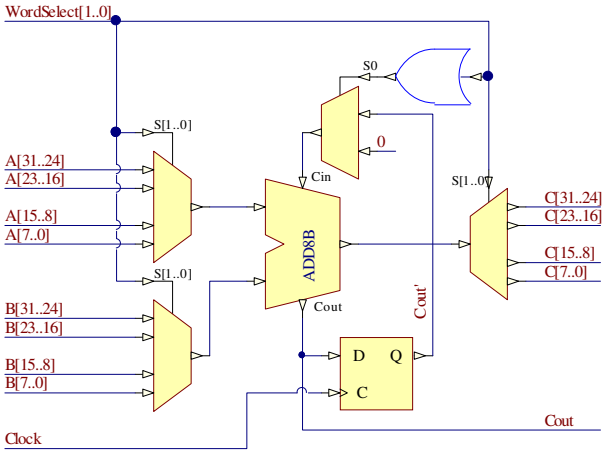


Figure 5 Performing 32-bit addition using a single, 8-bit adder

Continued extrapolation reveals that a 32-bit addition function could equally be performed with 32, single-bit additions but this, of course, would take around 32 times as long to execute as a single 32-bit addition.

Based on the above example, one might deduce that performance is approximately inversely proportional to the bit-width of the adder unit:

$$Performance \propto \frac{1}{bitwidth}$$

This assumes, however, that the machine performing these additions is able to easily access and reassemble the inner words of the A, B and C arguments and that the carry bit can be readily propagated across successive additions. The silicon resources required to split and reassemble the arguments may in fact be more than the savings made through reducing the word width of the adder component.

Clearly the performance versus bit-width relationship is non-linear and must take into account many more variables than are presented in this overly simplistic example. When applied to the challenge of developing a complete application, consequential effects must also be considered as the impact of changes made in one area of the application need to be considered in the light of the application's overall requirements.

By way of a further example, consider a processor that does not contain a hardware multiply unit. In this instance, multiplication may be implemented in software via a shift-add algorithm such as Booth's multiplication algorithm. Any changes made to the bit-width of the processor's adder unit will therefore have a flow-on effect to the resources available to any multiplications that are also to be performed. Conversely, making a hardware multiply resource available to the system would necessitate the inclusion of an adder unit that would then be available for use in other functions.

The impact that hardware resources have on software performance is the focus of ultra fine-grained hardware/software codesign. Given the complex nature of the hardware/software interactions it is necessary to employ some form of computer assistance with the partitioning decision for all but the most trivial applications. Ideally, this would allow alternate partitioning decisions to be evaluated and provide some direction as to which partitioning choices would produce a favourable outcome.

Representing Functional Requirements.

When the end goal is to produce a system that contains a mixture of both hardware and software, there exists the question of how the system level functionality should be represented; should it be represented using a *software* language, a *hardware* language, or some other alternative?

Using a *Software* Language to Represent Functional Requirements.

Where a software language is used to represent system functionality, the C language is often used as a starting point [101]. Typically the system engineer would begin by developing the code for the core application and prove its logical correctness on a PC or similar computing platform. At this point the code would most likely be profiled with a view to identifying tardy procedures or basic blocks that could be ported to a hardware implementation. For simple applications the porting process can be undertaken manually however for larger systems it is desirable to employ some level of automation.

The work presented in [102, 103] describes a 'C to VHDL' translation tool that is capable of converting a number of C constructs to VHDL. In a commercial context, Handel-C from Celoxica [104] has been in operation for over a decade producing tools which target C-based designs to FPGAs. Handel-C makes use of a number of language extensions to allow C, which is a sequential language, to represent hardware parallelism. SystemC, championed by Synopsys is an alternative option [105]. Whereas Handel-C allows the developer to

implement code with little consideration for system structure, SystemC enforces structural implementation to the extent that system-level functionality must be encapsulated into SystemC modules. Once the system is complete, the SystemC modules can be ported directly to hardware or left in software.

These two approaches vary in their advantages. Handel-C allows for system level development to proceed with little regard for final implementation specifics. Software can be tested on a generic platform using familiar tools. SystemC on the other hand produces a compilable output that, when run, constitutes a complete timing accurate simulation that can be used as a reference *executable specification*. Simulations run with SystemC are considerably faster than their VHDL counterparts and allow for verification of software and hardware.

Unfortunately significant limitations still exist with the Handel-C and SystemC approaches when considering processor development. Both these tools employ an *all-or-nothing* conversion approach and they are unable to partially transfer functionality to software without significant manual intervention and rewriting. Furthermore the tools don't have any mechanism to identify how hardware blocks can be utilized across the rest of the application. Any use of instantiated hardware is under the implicit direction of the user.

Using a *Hardware* Language to Represent Functional Requirements.

One significant limitation of using a software language to represent hardware is the fact that most software languages assume sequential execution and they don't have the constructs within them to support the level of true concurrency found in hardware. One might therefore consider using a *hardware* language as the starting point and convert code fragments to software. VHDL is a common hardware language and has been used in this approach [106]. Although the use of VHDL might be useful at higher abstraction levels, using it for functional level software development quickly reveals its inadequacies. VHDL does not support the concept of pointers and indirection and so performing common software tasks such as linked lists or record manipulations is quite challenging.

Ultimately there is a very good reason why both hardware *and* software languages exist; they are designed to serve distinct purposes. Trying to cajole one language form to represent constructs from a different domain fails to recognize their primary purposes.

Using a *New* Language to Represent Functional Requirements.

The limitations of using an existing hardware or software language to represent a conjoined system leads to one final option; use a *new* language that is equally capable of representing

both hardware and software. This proposition was considered very carefully during the early phase of this research program and would have been supported by arguments from the literature [107]. Page, along with other members of the Hardware Compilation Group at Oxford University's Computing Laboratory, used elements from the Occam language [108] to construct a new language called Handel. This language was fundamentally a software language but included all of the necessary constructs to represent hardware concurrency. It was, in fact, this language that formed the basis of the Handel-C product discussed above.
One might conclude that if there was merit in pursuing a new language in its own right then Page would have continued development with Handel and the language would still be alive today. In any case, Celoxica recognized that if what Handel had to offer was to be accepted by the broader community, it needed to *look* familiar. A comparison of the original Handel language syntax with its current Handel-C incarnation reveals that the original syntactic notation has been replaced by standard C notation. This can be interpreted as evidence that the acceptance of a codesign language will be determined by the broader design community's willingness to embrace change. Keeping the language fundamentally the same as such a well known language as C will help to ease acceptance. Further weight is given to this argument with the existence of SystemC as it also augments the familiar C language to accommodate hardware constructs.

An alternative to using a hardware, software or even new implementation language would be to use one of the various modelling languages proposed in the literature. Whilst these languages serve a useful system-level purpose, they face several challenges. The biggest limitation is the level of detail that they are able to contain. Modelling languages are purpose built to allow a designer to *model* a system's behaviour without the designer needing to focus on the specific details of the system they are modelling. This is helpful at early design stages where the system is being scoped however modelling languages are soon found lacking as the design process moves to more detailed phases. The flow of information between the abstract modelling languages and the detailed implementation languages is often lacking as changes can not be automatically propagated between the abstraction levels. Modelling languages serve their purpose at a systems level but are not suitable for use during implementation of conjoint systems.

Concluding remarks on Representing Functional Requirements.

The above discussion of Functional Representation leads me to the following preliminary conclusions:

1. It does not seem prudent to construct a new language
2. To increase the likelihood of broad acceptance, a new codesign method should be packaged into a familiar form or language.
3. The existing SystemC and Handel-C languages do not go far enough in their support for ultra-fine grained partitioning.

The Pivotal Role of the Compiler

The previous discussion focussed on how the functionality of a system should be represented however this is of little consequence if a compiler does not exist that is capable of taking that functional representation and mapping it onto the available hardware. As previously mentioned in Chapter 2, a study into the relative effects of architectural change versus compiler optimization techniques revealed that the compiler had a far greater effect on overall performance than architectural changes alone [58].

If we accept that the purpose of a compiler is to map a functional description onto the available hardware resources then it seems reasonable to assert that the compiler is extremely well positioned to play a pivotal role in future codesign. Irrespective of the amount of hardware or software that is eventually used when implementing a conjoint system, the compiler sits at the intersection point where hardware and software collide. It *knows* about the required system functionality and it also *knows* about the available hardware resource.

Compiler technology and the construction of compilers is a field that spans back to the early beginnings of software in the late 1950s [5]. However even with the existence of a range of tools that assist with compiler development, it is currently estimated that retargeting the GNU Compiler Collection (GCC) to a new processor takes about 3-4 man-months for someone who knows what they are doing [109]. Whilst a number of processors have already been targeted and provide developers with the opportunity to try their code on a range of processors, if a developer wishes to modify the processor core or test their code on a fictitious processor then the development lead times are totally prohibitive.

Some work has been done to decrease targeting time through tools such as LISA [110] and AVIV [111] as these tools enable the designer to quickly build compilers and simulators for a new target architecture. They appear to be quite useful for exploring different architectural

alternatives but they don't provide a starting point for architectural development. Their emphasis is on enabling the rapid representation of a pre-existing hardware architecture rather than on determining what hardware architecture is best suited for a given application. This work is important but it does not really provide a starting point for developing a custom processor nor does it provide an indication of the value that the compiler might place on having certain hardware resources available to it.

Compiling for a Fictitious Processor

By extending the idea of using the FPGA as the processor platform, it becomes apparent that the choice of processor core is totally flexible. In this environment, instruction *extension* is not really a consideration as there is no concrete processor to extend. A more important question is "what processor architecture should be used for a given application?". When existing processor constraints are removed, knowing where to begin building an entirely custom unit is quite daunting. Developers face the proverbial chicken or egg dilemma since the hardware needs are dependant on the software but the compilation of the software is dependant on the hardware.

This problem is one that ASIP developers have faced for some time. Drawing from their work are several examples where a base architecture is assumed and instruction extensions are added from there [74, 112-116]. In general, this research involves determining which instruction *extensions* are best to include as opposed to determining the *core* instructions and architecture.

In truth, some architectural line in the sand needs to be drawn before it is possible to begin instruction selection. A premise of the work in this thesis has been that the initial architectural model should be as abstract as possible and not based around an underlying hardware implementation that could prejudice the final outcome. For instance, work done on the BUILDABONG [117] project is highly relevant to this work however they base their designs upon an existing VLIW processor. Their goal appears centred around DSP algorithms where instruction level parallelism is both possible and desirable however this computing model is not as relevant in other domains. For example, it is more difficult to draw out parallelism in control oriented applications such as typically found in automotive applications and many of the benefits of a VLIW processor are lost.

In Chapter 2 I have already emphasized the importance of structure being made subservient to function and so in order to maintain generality, representations of the computing machine

must be as abstract as possible. Structural boundaries should only be constructed once a clear understanding of the application's functional requirements have been established. CISC, RISC or VLIW architectural structures can then all be explored. Making assumptions about the architectural composition too early can lead to an overly restricted and sub-optimal implementation.

Other Closely Aligned Work

The PEAS Project

The PEAS (Practical Environment for ASIP Development) project is very closely aligned with the work in this thesis. Its development extended for over a decade and resulted in a number of papers [46, 118-120]. The primary focus of this work is to identify the *best* instruction set, based on area/performance constraints, for a given application. Although their work targets ASIPs, it is just as relevant in targeting modern day FPGAs. PEAS is comprised of four subsystems;

1. APA: Application Program Analyzer
2. AIG: Architecture Information Generator
3. CCG: CPU Core Generator, and
4. DTG: Application Program Development Tool Generator.

The APA subsystem uses a profiling method to determine the relative usage of different instruction types. This information is then passed to the AIG subsystem which uses a database to determine which hardware modules should be used to support the instruction types. The hardware modules are characterized by their area and the number of clock cycles they take to execute their given function. A branch and bound optimisation technique is used to select the most appropriate combination of hardware modules given a specific performance/area constraint. The CCG and DTG subsystems take the hardware module set determined by the AIG subsystem and automates the production of the CPU core and development tools respectively.

PEAS makes use of GCC's (GNU C Compiler) intermediate RTL representation to determine the relative usage of different instruction types. These RTL instructions are classified as Primitive RTL (PRTL), Basic/Simple RTL (BRTL/SRTL) and Extended RTL (XRTL). The assumption is made that all PRTL instructions will be implemented in hardware whilst different BRTL and XRTL instructions may be implemented entirely in

hardware or as a mix of hardware and software. The exact mix is determined by the APA subsystem.

The developers of PEAS advocate its ability to rapidly converge on a candidate hardware set. In 1992 they were reporting less than 1 second to converge using a SUN-3/60 (3 MIPS) workstation [118]. The speed of this approach however, was not without its limitations. Execution time predictions were found to have a worse case error rate of between 15% - 35% for a set of test applications [121]. This error rate was most noticeable under high software subroutine usage (low hardware). The reason they offer is that the execution cycles consumed by the software subroutines was data dependant. Unsurprisingly, the performance prediction error gradually decreased as more of the functionality was implemented in hardware. By 1996, the PEAS developers had addressed this prediction error through the use of an adaptive database approach and were reporting error rates below 2% [119].

Based on the published convergence rates and prediction errors, it would appear that the PEAS project used GCC to make an initial pass of the target application from which the intermediate RTL representation could be extracted and a solution calculated. The fact that large (data dependent) error rates were reported in their initial findings is consistent with the likelihood that they were not adequately taking into account the increased register spilling that occurs when less hardware resources are available. The use of a predictive algorithm to reduce the magnitude of the error rates is an effective remedial measure but does not address the root source of the errors.

This thesis proposes a slightly different approach to that of the PEAS project. By changing the codesign process to an iterative one that includes the compiler in the design loop and testing each hardware/software solution along the way, the need to make predictions can be totally removed since the compiler output can be measured directly. The benefit is absolute accuracy in the performance predictions but at the cost of slower convergence to a solution.

Tensilica

Tensilica has achieved significant maturity in the development of configurable ASIPs with their commercially available Xtensa configurable processor core. Both the process of configuring this core [43, 45, 122] and the benefits [44, 74] have been published in the literature. Using Tensilica's Instruction Extension (TIE) language, designers can describe extensions to the base ISA and have those extensions realized in hardware and recognized by the software tool chain. Once a processor has been configured, designers can log in to Tensilica's internet-based generation process to produce the processor's configured RTL

description along with its associated software development tools including an ANSI C/C++ compiler, linker, assembler, debugger, code profiler, and instruction set simulator [43].

Tensilica's automated instruction extension tool, AutoTIE, uses a mixture of compiler derived information along with area estimates to automatically extend a base processor with application specific register files, operations, and instructions that are automatically used by the software tool chain [45]. Specifically, it considers combinations of VLIW Operations, Vector Operations, and Fused Operations (the merging of several atomic instructions into a single instruction) in order to achieve maximum speedup under constrained area requirements.

In spite of the remarkable success that Tensilica has achieved, there are still several differences between their approach and my own as well as opportunities for further research. Tensilica assumes a fixed 32-bit base processor architecture that is to be extended through their configuration process. This is ideally suited to embedded applications that have demanding performance, size, and power requirements that cannot be satisfied by a general purpose processor [45]. This thesis explores the opposite end of that spectrum where minimal area is the goal rather than speed alone. So rather than adding hardware to achieve speedup, this thesis explores the process by which processor hardware can be removed whilst still achieving a minimal level of performance. But as observed by [52], ASIP design solutions that use a base processor instruction set that cannot be pruned do not lend themselves well to finding minimal area solutions.

Furthermore, Tensilica have focussed on custom ASIP development for developers who are interested in producing their processors in hard silicon. So even though the RTL produced by the generation process can be verified on an FPGA-based platform, the resources consumed by this type of deployment make them prohibitive for use in FPGAs. It has been shown [123, 124] that variants of the Xtensa-V configurable processor consume between 12,332 and 29,622 Look Up Tables (LUTs) on a Virtex2 FPGA. By comparison, Xilinx's hand crafted 32-bit MicroBlaze core running on equivalent hardware (Spartan3) requires as little as 1324 LUTs [125]. In addition, while Xtensa-V can only manage 29-33MHz when run on an FPGA, the Microblaze can run at up to 115MHz on equivalent hardware.

If we accept the assertion that the proliferation of reconfigurable systems is inevitable [13], then it is clear that minimal area reconfigurable solutions will as important to some designers as maximum performance is to others. It is this notion that has motivated the work in this thesis.

Concluding Remarks

This chapter has provided a more in-depth description of the codesign problem and has shown some of the limitations of other work. The emergence of high capacity, low-cost FPGAs enables total control over the underlying hardware and removes any restrictions on processor structure and composition. This presents an interesting challenge of how one should approach codesign given such hardware freedoms. Whereas the bulk of existing approaches have assumed that some portion of the hardware platform is fixed and the software and compilers have been developed with that in mind, the premise of this thesis is that the application should be fixed and the compiler and hardware should develop around that.

The PEAS project has used GCC's intermediate language to create an initial RTL representation that can be used as the starting point for their analysis but they do not include the compiler at any other point in the codesign process. The work reported by this thesis in the following chapters will extend beyond what has previously been published by others and will include the compiler as an integral part of the codesign process. Currently, it is a slow technique that involves several iterative steps and at each step the entire compiler needs to be rebuilt to include the hardware modifications made as part of the previous iteration. However the benefit of this technique is that there is no reliance on predictive calculations since the impact of each design decision can be directly measured from the compiler output. Once the compiler refinement process is complete, a functional hardware specification can be drawn from the compiler's machine description file and then be used to direct the hardware development and subsequently produce a working platform.

Chapter 4.

LAYING THE FOUNDATIONS FOR A COMPILER DIRECTED CODESIGN PLATFORM

Having examined the literature in Chapter 2, and through an understanding of the alternate approaches to codesign presented in Chapter 3, we are now in a position to examine in detail the expanded use of a compiler in the codesign process. Compilers are uniquely positioned at the pivotal point between hardware and software; they are aware of an application's functional requirements as well as the available hardware resources upon which the application is to run. In an abstract sense the compiler is already responsible for taking a high-level application description and mapping it onto a hardware/software platform – i.e. a processor that executes assembly instructions, albeit assuming a fixed target micro-architecture.

This chapter will outline the attributes that must be present in a compiler for it to be used as part of a Compiler Directed Codesign system. I will then present the compiler that was chosen as the basis for the Compiler Directed Codesign platform and describe how that compiler can be retargeted to a new processor. The final section of the chapter will show, via experimentation, how some of the key compiler features can be used in developing the target assembly instructions and how some of the limitations of the compiler that were exposed by this methodology can be assessed and overcome.

An Overview of the Compiler Directed Codesign Methodology

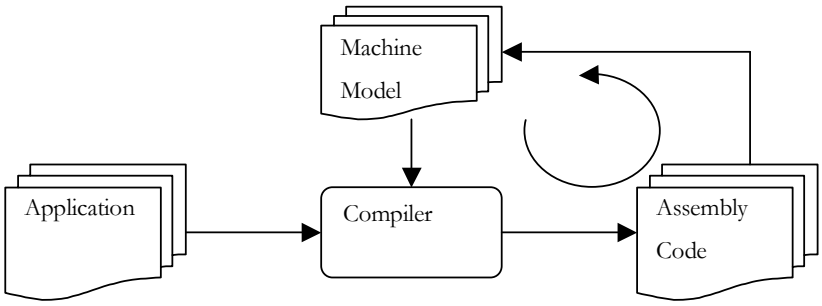


Figure 6 Outline of the Compiler Directed Codesign workflow.

The Compiler Directed Codesign approach broadly involves three steps that applied iteratively, will allow the designer to converge on a codesign solution (Figure 6).

1. Compile the application.
2. Observe the assembly code generated by the compiler.
3. Adjust the compiler's machine model based on information in step 2. Rebuild the Compiler and start the process again from step 1.

Compiler Requirements

In order for this process to be used, it is essential that key aspects of the compiler's behavior can be controlled and their effects observed. When determining a compiler's suitability, there are a number of attributes which are important.

Retargetability

The first and foremost requirement is retargetability. As the target processor is not known from the outset, the compiler must be sufficiently flexible to accommodate an evolving hardware model. Ultimately one might argue that given sufficient time, any compiler could be redeveloped for a new hardware platform however since the Compiler Directed Codesign method involves repeated test and measure cycles, it is particularly desirable that the compiler is *rapidly* retargetable.

Optimizing Compiler

The second primary consideration is the optimality of the code produced by the compiler. Since the compiler can have a far greater impact on code performance than hardware resources alone [58], it is essential that sufficiently complex optimization strategies are employed by the compiler to maximize code efficiency.

Code 'Morphing' Capability

The ability to merge or split instructions of one form into another is also highly desirable. Most compilers are capable of mapping complex operations to relatively simple, atomic assembly statements. However, conjoined designs may seek to make use of additional hardware to accommodate highly specific computational sequences. The compiler needs to not only reduce complex sequences into atomic instructions, but also be sufficiently flexible to know how to merge or morph instruction sequences into a superset form that makes maximum use of the specialized hardware resources.

Economic/Documentation Factors

Although these attributes have minimal technical impact, they are an important consideration given the resources available to this research endeavour. Compilers have evolved to be extremely complex pieces of software. Since the proposed methodology may involve some redevelopment of the compiler internals, it is desirable that sufficient documentation exist to help inform that process. Furthermore it is desirable that acquiring access to the compiler's source code involves minimal cost (i.e. free).

The GNU Compiler Collection (GCC)

There are many open source and free compilers readily available online. These compilers cover languages such as Pascal [126, 127], C/C++ [9, 128-130], COBOL[131], and Java [132, 133]. The focus of this thesis however is not concerned with the language (the compiler front end) but rather the means through which the compiler targets an underlying instruction set architecture (the compiler back end).

Having assessed the compiler requirements as they pertain to Compiler Directed Codesign, I concluded that the GNU Compiler Collection (GCC) presented as an excellent candidate and would serve as a suitable base for a Compiler Directed Codesign platform.

GCC is an open source compiler suite that can be freely downloaded from the GCC home page [128]. It has been purpose-built for retargetability and has been ported to more kinds of processors and operating systems than any other compiler [134]. It is supported by a reasonable level of documentation [135] and an active community of developers. GCC possesses all of the attributes deemed important for supporting Compiler Directed Codesign. Once installed, GCC supports around 60 host/target platforms. Retargeting the compiler to a new processor involves the creation of a number of description files specific to the target architecture and linking those files into the compiler during a compiler build.

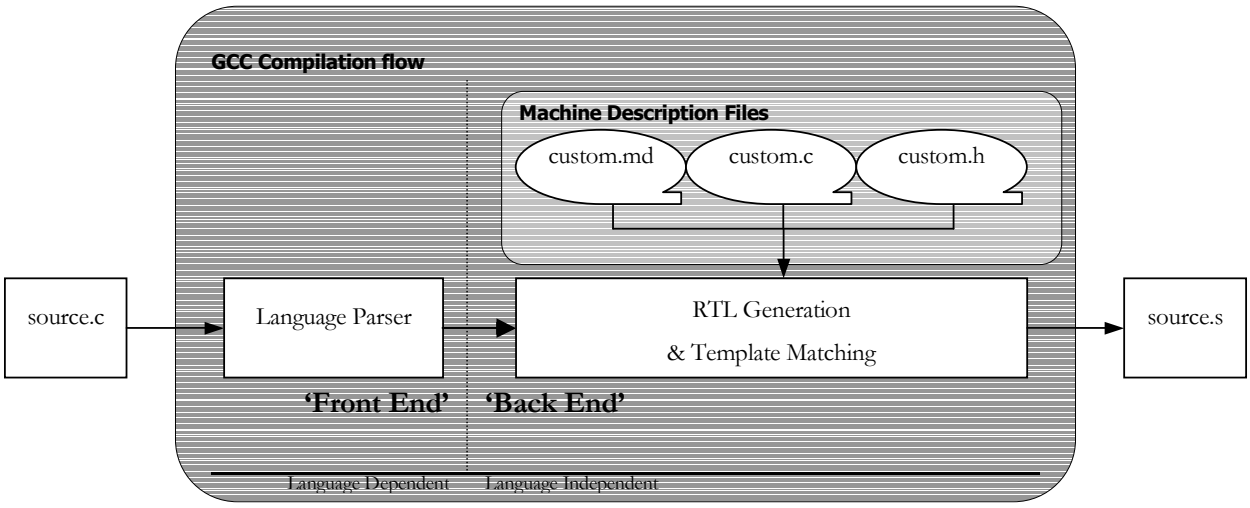


Figure 7 Simplified View of the GCC Compiler flow

GCC Structure

As part of the compilation flow, there are three main conversion phases that happen in the compiler (Figure 7):

1. The front end reads the source code and builds a parse tree.

   The *front end* is responsible for managing those aspects of the compilation process that depend primarily on the source language and are independent of the target machine. Output of the *front end* is a tree-like representation of the application code with properties similar to that of a Directed Acyclic Graph (DAG). This work does not involve modifications to the front end of GCC and it can be assumed to be a black box.

2. The parse tree is used to generate an RTL *insn*[2] list based on named instruction patterns.

   GCC defines a set of 118 RTL instruction patterns that are hard-coded into the compiler. Each has a unique name that identifies its operation. When creating the initial *insn* list, the compiler will search the `custom.md` file for an RTL template with the name corresponding to the operation being performed in the parse tree. If found, the RTL template will be inserted into the *insn* list and processing of the parse tree will continue. If the named template can not be found – i.e. the compiler seeks to use a standard instruction pattern which has not been defined for the target architecture – then the

[2] The term *insn* appears regularly throughout the GCC documentation and although no formal definition of the term is offered, a post on the GCC forum suggested that it is an abbreviation of the word *instruction*. Taken in context, *insn* refers to the intermediate instructions used internally by GCC to represent RTL operations.

compiler will attempt to use a different strategy involving other instruction patterns. If all alternate strategies are exhausted, the compiler will abort with an error message.

A number of optional optimization passes may follow this step along with *insn* reorganization and register spill management.

3. The *insn* list is matched against the RTL templates to produce assembler code.

   The final phase of the compiler is to again use the RTL templates defined in `custom.md` to match against the completed *insn* list. Where a match is found, the portions of assembly code associated with the matching templates are inserted into the assembly output to complete the compilation process.

Adding support to GCC for a new or custom processor is handled by three files; `custom.md`, `custom.c`, and `custom.h` where *custom* represents the name of the target processor that is used to distinguish it from other ports.

custom.md

The `custom.md` file is the Machine Description (md) file. Using a template structure, the '`.md`' file contains an *insn* pattern for each instruction that the target machine supports. The compiler uses standard named patterns as well as target specific patterns to create and process an *insn* list. A number of optimization passes may modify and refine the *insn* list during compilation but once complete, the *insn* list will be translated into target assembly statements according to information contained in the *insn* templates. *Insn* list construction and modification can optionally make use of two other 'C' files that contain further information about the machine's attributes.

custom.h

`custom.h` contains a number of macros which are used by the compiler to describe static aspects of the target architecture i.e. word size, endianness, calling conventions, size/type of available registers and their class, byte alignment, etc. Macro definitions can be static i.e.

```
#define POINTER_SIZE 16
```

or make use of a C function call to calculate its result

```
#define POINTER_SIZE get_machine_pointer_size();
```

If a C function is used it must be defined in `custom.c` (Figure 8).

custom.c

The `custom.c` file is a list of user-defined C functions that can be called from macros defined in `custom.h` or as part of back end processing of *insns*.

Figure 8 Example RTL Template

In order to represent all of the information necessary for back-end processing, the `custom.md` file must contain several pieces of information for each RTL template. An example template is given in Figure 8. The key items of an RTL template are as follows:

Template Name

The Template Name is given on the first line in quotes. Only standard RTL templates need have a name as without one, the compiler will not be able to make use of them during initial RTL generation. The example above shows a template for a 3-operand add instruction of mode 'QI' (Quarter Integer = 8-bits). The name of this RTL template is `addqi3`.

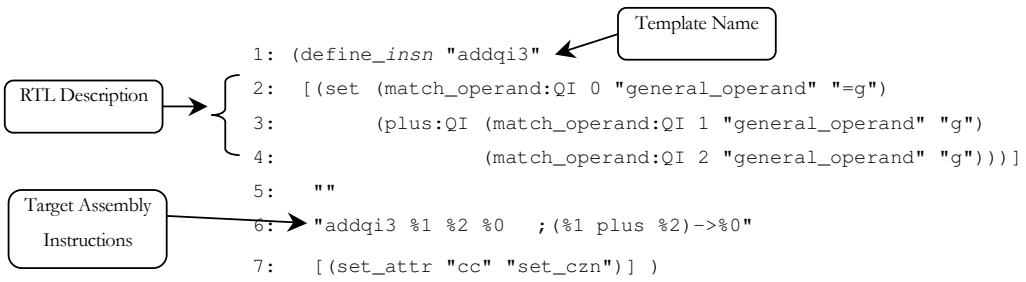
RTL Template Description

The RTL Template Description is bound by [..] and defined on lines 2 to 4. In this example it states that operand 0 of type QI will be set with the result of the addition (plus) of operands 1 and 2 (both of type QI). The "`general_operand`" and "`=g`" specify constraints that must be satisfied by the operands during the RTL matching pass. In the this case the operands can come from registers or memory.

Target Assembly Instructions

The Target Assembly Instructions on line 6 contains a string of target-specific assembly instructions that will perform the function of the RTL Template Descriptor.

The Template Name and RTL Template Description are used during the RTL Template



Generation pass of the compiler. The RTL Matching and Instruction Generation pass uses the RTL Template Description when searching the RTL List for valid patterns and, once found, inserts the Target Assembly Instructions into the assembly output listing of the compiler.

Attribute Data

The information on line 7 contains optional attribute data that describes the impact that the Target Assembly Instructions will have on the machine's condition codes. Other, user defined attributes can also be set at this point.

Expanding RTL Templates

The `define_insn` descriptor is given to an RTL Template when it is possible to directly translate the RTL description into the target machine's instruction set. Where direct translation is not supported by the target instruction set, a single, standard RTL Template can be expanded into a sequence of RTL Templates that do support direct translation. In this situation, GCC offers a `define_expand` descriptor, outlined in Figure 9, below:

```
(define_expand "addhi3"
[(set (match_operand:HI 0 "general_operand" "=g")
      (plus:HI (match_operand:HI 1 "general_operand" "g")
               (match_operand:HI 2 "general_operand" "g")))]
 ""
 "{
  operands[3] = gen_reg_rtx (QImode);

  emit_insn(gen_addqi3 (custom_subword(operands[0], 0, HImode),
                        custom_subword(operands[1], 0, HImode),
                        custom_subword(operands[2], 0, HImode) ));
  emit_insn(gen_cmpqi (custom_subword(operands[0], 0, HImode),
                       custom_subword(operands[1], 0, HImode) ));
  emit_insn(gen_sltu (operands[3]));
  emit_insn(gen_addqi3 (custom_subword(operands[0], 1, HImode),
                        custom_subword(operands[1], 1, HImode),
                        custom_subword(operands[2], 1, HImode) ));
  emit_insn(gen_addqi3 (custom_subword(operands[0], 1, HImode),
                        operands[3],
                        custom_subword(operands[0], 1, HImode) ));
  DONE;
  }" )
```

Figure 9 Example of `define_expand` template

These `define_expand` templates can only be used to support standard RTL Templates since they are accessed by name. The template in Figure 9 expands a single 16-bit add into a number of 8-bit adds along with some carry propagation instructions. The assembly output derived from this `define_expand` template is as follows:

```
            addqi3 ([b1+#3]) ([b2+#3]) (r28+#4)
            cmpqi (r28+#4) ([b1+#3])
            sltu_qi r24
            addqi3 ([b1+#2]) ([b2+#2]) (r28+#3)
            addqi3 r24 (r28+#3) (r28+#3)
```

Figure 10 Assembly output derived from `define_expand` template

The existence of the `define_expand` descriptor is significant to codesign applications because it provides a mechanism to strictly control how complex RTL Templates can be expanded into a sequence of less complex (atomic) templates. Without this feature, *insn*

reduction would only be possible via hard-coded algorithms embedded within GCC. Although they will produce correct code, they are generalized algorithms designed to support a range of processor types and are not easily changed to suit custom organizations.

For example, the `define_expand` template shown in Figure 9 expands the standard 16-bit addition template (`addhi3`) into a sequence of 8-bit additions and was derived from one of the GCC expansion algorithms. One might expect from this example that a similarly formed expansion could translate a single 32-bit addition into a sequence of 16-bit additions. Unfortunately, GCC is not capable of doing this expansion internally for an 8-bit machine. GCC will first look for the availability of a standard 32-bit addition *insn* (`addsi3`). If this is not available, GCC will then implement the addition function using the target machine's base word size. In the case of an 8-bit processor, GCC will emit a sequence of 8-bit addition *insns* (`addqi3`) **even if a standard 16-bit addition insn (addhi3) has been defined**. Figure 11 illustrates this point.

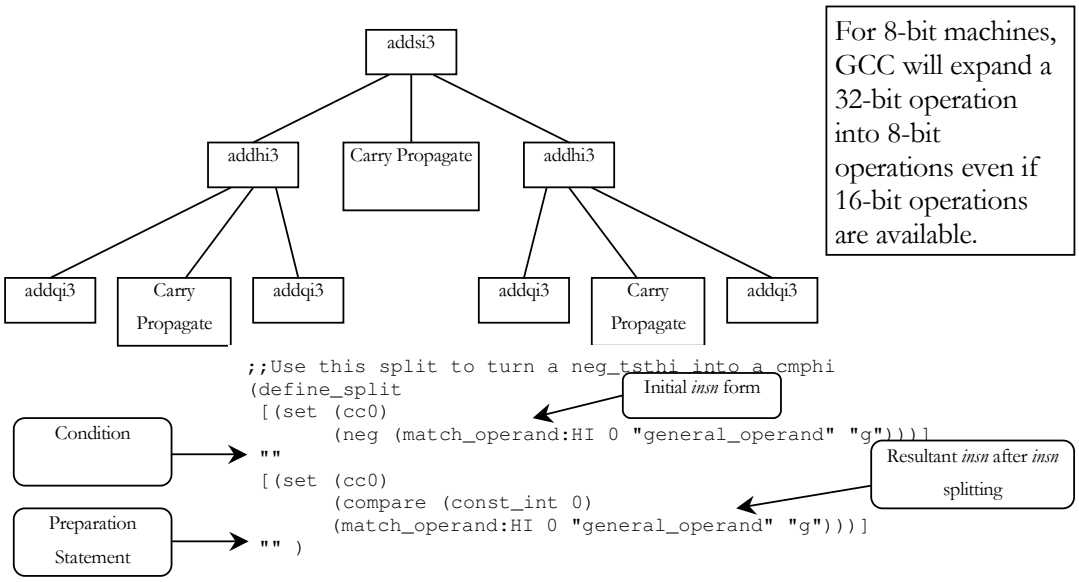


Figure 11 Expansion of 32-bit addition into 2 x 16-bit additions and 4 x 8-bit additions

By using the `define_expand` feature, users can override GCC's default behaviour and explicitly control how larger word operations are expanded; i.e. the 32-bit add can be expanded into two 16-bit additions rather than four 8-bit additions.

Figure 12 Insn splitting with define_split.

Insn Expansion with `define_split`.

As previously mentioned, the `define_expand` descriptor is only useful with standard, named RTL templates because the compiler references them by name. In some circumstances,

however, it is desirable to expand or split other, non-named *insn* patterns into another form. For these situations GCC provides the `define_split` descriptor. A simple example of using the `define_split` descriptor is given in Figure 12.

One of GCC's named *insn* patterns is called `tstm`. The role of this *insn* is to set the machine's condition codes based on the argument provided with the `tstm` *insn*. In some circumstances the compiler will decide to use a negative form of the `tstm` *insn* and will produce the *insn* pattern listed in the first portion of Figure 12 (see 'initial *insn* form'). This *insn* pattern may not be supported on some machines and so it is necessary to modify the *insn* pattern to another form that is functionally equivalent. In this case, the `neg_tstm` *insn* will be modified to the form of a compare *insn* (see 'Resultant *insn* after *insn* splitting').

Judicious use of `define_expand` and `define_split` *insn* patterns can offer a very high degree of control over how *insn* patterns are expanded into smaller, atomic *insns*. In essence, these two descriptors allow the system designer to explicitly control how a larger word-size algorithm can be computed on a smaller word-size machine. The compliments of these two templates are *insn* templates that enable the designer to merge multiple *insns* together into one, or more, 'super' *insns*. These are described in the following section.

Contracting RTL Templates

One of the primary building blocks of DSP algorithms is the multiply-accumulate (MAC) instruction sequence. In response to this regular instruction pattern, virtually all DSP processors contain a MAC instruction. GCC does not contain a standard RTL template for MAC instructions. Instead, a RTL template needs to be defined which merges two related multiply and addition operations into a single *insn*.

A similar problem exists with the carry propagate logic shown in Figure 11. Currently GCC does not distinguish between additions made with or without a carry bit and so it needs to use the lengthy sequences shown in Figure 9 and Figure 10 to propagate a carry. More efficient code could be implemented if the sequence of five instructions could be reduced to just two addition instructions that contained the necessary carry propagation logic.

GCC provides two mechanisms for instruction reduction. The first mechanism is employed at the assembly output stage and is used to contract a sequence of *insns* into one or more assembly instructions. Templates defined for use during this phase are denoted by the `define_peephole` descriptor. An example `define_peephole` template that deals with the carry propagation problem is given below (Figure 13):

```
;;Peephole to merge 2 addhi's into an addhi & addchi
(define_peephole
 [(set (match_operand:HI 0 "general_operand" "")
           (plus:HI (match_operand:HI 1 "general_operand" "")
                    (match_operand:HI 2 "general_operand" "")) )
  (set (cc0)
       (compare (match_dup 0) (match_dup 1)) )
  (set (match_operand:VOID 3 "general_operand" "")
       (ltu:VOID (cc0) (const_int 0)) )
  (set (match_dup 3)
       (plus:HI (match_dup 3)
                (match_operand:HI 4 "general_operand" "")) )
  (set (match_dup 3)
       (plus:HI (match_dup 3)
                (match_operand:HI 5 "general_operand" "")) )]

 ""
 "addhi3 %1 %2 %0
  \taddchi3 %4 %5 %3" )
```

Figure 13 Peephole template for reducing a long *insn* sequence into a relatively short assembly instruction sequence.

The template of Figure 13 will produce an assembly instruction sequence similar to Figure 14 when matched. In this example, the first assembly line is for a 16-bit addition (with no input carry) and the second assembly line performs a further 16-bit addition but with regard to the state of the carry flag – i.e. any carry generated from the first instruction will be propagated into the second instruction.

```
addhi3 r21 r23 r25
addchi3 r20 r22 r24
```

Figure 14 Assembly code output as a result of peephole reduction performed using template in Figure 13.

It can be argued that it would be far more efficient to output the assembly code of Figure 14 directly from an initial addsi3 *insn* template and in this simple example this would be correct. The purpose of this discussion however is to demonstrate the flexibility that *insn* expansion and reduction affords and to show how this mechanism can be used to create new, unique instructions that can be tuned for a specific application.

Insn Reduction with `define_peephole2`.

The second instruction reduction technique available is very similar to the previous example however instead of emitting assembly code, it emits an *insn* sequence which is then available for further instruction matching or reduction. Templates designed for this form of *insn* reduction are denoted as `define_peephole2`. They are very similar in structure to the `define_split` template however the pattern to match is not a single *insn* but rather a sequence of *insns*. A simple example of *insn* reduction using this mechanism can be seen in Figure 15.

```
;; Peephole to turn 2 movqi's into a single movhi
(define_peephole2
 [(set (match_operand:QI 0 "general_operand" "")
       (match_operand:QI 1 "general_operand" "") )
  (set (match_operand:QI 2 "general_operand" "")
       (match_operand:QI 3 "general_operand" "") )]
 "custom_check_adjacent(operands[1], operands[3])
           && custom_check_adjacent(operands[0], operands[2])"
 [(set (match_dup 4) (match_dup 5))]
 "operands[4] = gen_rtx (GET_CODE(operands[0]), HImode, XEXP(operands[0],0));
  operands[5] = gen_rtx (GET_CODE(operands[1]), HImode, XEXP(operands[1],0)); ")
```

Figure 15 Peephole template for replacing an *insn* sequence with another *insn*.

The example in Figure 15 reduces two 8-bit data moves into a single 16-bit data move. `custom_check_adjacent()` is a function that checks that the operands passed to it are adjacent to one another and can therefore be considered as part of a single, larger-word operand. The form of the substituting *insn* is provided on the third last line and operand widening is effectively done on the final two lines. The `define_peephole2` template only operates on *insns* however its effect can be seen below. A machine which does not support the `movhi` *insn* will emit assembly code with the following form to perform a 2-byte move of data in the register pair r21:r20 to the new location r22:r23.

```
movqi r21 r23
movqi r20 r22
```

Figure 16 Performing a two-byte move on a machine without a movhi *insn* defined.

By employing the substituting template of Figure 15, the previous two lines of assembly code could be reduced to this single, 16-bit move instruction:

```
movhi r20 r22
```

Figure 17 Using Peephole optimization to reduce a two-byte move to a single instruction.

Understanding and Overcoming GCC's Limitations

The template expansion and reduction mechanisms described so far demonstrate the features of GCC that make it well suited for Compiler Directed Codesign purposes but these capabilities are not without their limitations. The rigid ordering of each of the compiler's optimisation passes means that successive expansion and reduction phases are generally not possible without some modifications being made to the compiler core. These limitations are examined and discussed in the remainder of this chapter.

Examining GCC Limitations

A series of simple test applications were developed to exercise GCC's various *insn* expansion and reduction mechanisms. They revolved around a simple program that performs a single

32-bit addition. GCC would normally seek to make use of an `addsi3` (32-bit add) *insn* to fulfil this request however in these experiments, only the `addhi3` (16-bit add) and `addqi3` (8-bit add) *insns* were made available. Various combinations of `define_insn`, `define_expand`, `define_peephole`, and `define_peephole2` were used to characterize the compiler. A summary of the tests are given below:

Source code

```
long int b1 = 1000, b2 = 1000;

long int
 main (void)
 {
   return(b1+b2);
 }
```

Figure 18 Test Case C program exhibiting a single 32-bit addition.

Notes: The `type` of a `long int` in this application is 32-bits wide.

Experiment 1

Aim: To observe which *insns* the compiler uses when given total freedom of choice.

Conditions: All *insns* of mode `SI` or smaller (≤ 32-bits) are available for use by the compiler. `addsi3` is enabled for registered operands.

```
(define_insn "addsi3"
 [(set (match_operand:SI 0 "register_operand" "=r")
       (plus:SI (match_operand:SI 1 "register_operand" "r")
                (match_operand:SI 2 "register_operand" "r")))]
  ""
  "addsi3 %1 %2 %0   ;(%1 plus %2)->%0" )
```

Figure 19 Extract of Machine Description file (`custom.md`) showing `addsi3` *insn* template.

Compiler Output[3]:

```
            addsi3 r24 r18 r24      ;(r24 plus r18)->r24
```

Figure 20 Extract of assembly file produced after compilation.

Discussion: As expected the compiler made use of the `addsi3` to fulfil the 32-bit add requirement.

[3] For brevity, compiler output is restricted to only include those lines of code that are relevant to the experiment's outcome.

Experiment 2

Aim: To observe which *insns* the compiler uses when its preferred *insn* is not available.

Conditions: The `addsi3` *insn* was removed from the compiler's machine description file and was therefore not available for use.

Compiler Output:

```
addqi3 r21 r17 r27      ;(r21 plus r17)->r27
cmpqi r27 r21     ;(r27-r21)->CC
sltu r22
addqi3 r20 r16 r26      ;(r20 plus r16)->r26
cmpqi r26 r20     ;(r26-r20)->CC
sltu r30
addqi3 r22 r26 r23      ;(r22 plus r26)->r23
cmpqi r23 r26     ;(r23-r26)->CC
sltu r22
iorqi3 r30 r22 r30      ;(r30 ior r22)->r30
movqi r23 r26     ;r23->r26
addqi3 r19 r15 r25      ;(r19 plus r15)->r25
cmpqi r25 r19     ;(r25-r19)->CC
sltu r23
addqi3 r30 r25 r30      ;(r30 plus r25)->r30
cmpqi r30 r25     ;(r30-r25)->CC
sltu r22
iorqi3 r23 r22 r23      ;(r23 ior r22)->r23
movqi r30 r25     ;r30->r25
addqi3 r18 r14 r24      ;(r18 plus r14)->r24
addqi3 r23 r24 r24      ;(r23 plus r24)->r24
```

Figure 21 Extract of assembly file produced after compilation.

Discussion: Even though the `addhi3` *insn* (16-bit addition) was available, the compiler defaulted to using the minimum word size that was defined for the processor. This, of course, is quite an inefficient way to perform the 32-bit addition.

Experiment 3

Aim: To observe that the compiler makes appropriate use of an expanded *insn*.

Conditions: Use a `define_expand` *insn* to explicitly tell the compiler how to expand the `addsi3` *insn*.

```
(define_expand "addsi3"
 [(set (match_operand:SI 0 "register_operand" "=r")
       (plus:SI (match_operand:SI 1 "register_operand" "g")
                (match_operand:SI 2 "register_operand" "g")))]
  ""
  "{
   operands[3] = gen_reg_rtx (QImode);
   operands[4] = gen_reg_rtx (HImode);

   emit_insn(gen_addhi3 (custom_subword(operands[0], 0, SImode),
                                        custom_subword(operands[1], 0, SImode),
                                        custom_subword(operands[2], 0, SImode) ));
   emit_insn(gen_cmphi ( custom_subword(operands[0], 0, SImode),
                                        custom_subword(operands[1], 0, SImode) ));
   emit_insn(gen_sltu (operands[3]));
   emit_insn(gen_zero_extendqihi2(operands[4],operands[3]));
   emit_insn(gen_addhi3 (custom_subword(operands[0], 1, SImode),
                                        custom_subword(operands[1], 1, SImode),
                                        custom_subword(operands[2], 1, SImode) ));
   emit_insn(gen_addhi3 (custom_subword(operands[0], 1, SImode),
                                        operands[4],
                                        custom_subword(operands[0], 1, SImode) ));
   DONE;
   }" )
```

Figure 22 Machine Description File showing the expanded form of the `addsi3` *insn*.

Compiler Output:

```
            addhi3 r20 r16 r26      ;(r20 plus r16)->r26
            cmphi r26 r20     ;(r26-r20)->CC
            sltu r22
            zero_extendqihi2 r22 r22      ;zero_extend(r22)->r22
            addhi3 r18 r14 r24      ;(r18 plus r14)->r24
            addhi3 r22 r24 r24      ;(r22 plus r24)->r24
```

Figure 23 Assembly code produced by compiler.

Discussion: The assembly output correctly reflects the code that would be expected from the *insn* expansion template.

Experiment 4

Aim: To observe the compiler's ability to reduce a complex *insn* sequence into a short assembly code sequence.

Conditions: Keeping the expanded form of `addsi3`, a `define_peephole` template was added to the machine description file with the purpose of translating a relatively long *insn* sequence into a shorter assembly sequence comprising of 'add' and 'add with carry' instructions.

```
(define_peephole
 [(set (match_operand:HI 0 "general_operand" "")
                      (plus:HI (match_operand:HI 1 "general_operand" "")
                               (match_operand:HI 2 "general_operand" "")) )
  (set (cc0)
       (compare (match_dup 0) (match_dup 1)) )
  (set (match_operand:QI 3 "general_operand" "")
       (ltu:QI (cc0) (const_int 0)) )
  (set (match_operand:HI 4 "general_operand" "")
       (zero_extend:HI (match_dup 3)))
  (set (match_operand:HI 5 "general_operand" "")
       (plus:HI (match_operand:HI 6 "general_operand" "")
                (match_operand:HI 7 "general_operand" "")) )
  (set (match_dup 5)
       (plus:HI (match_dup 4)
                (match_dup 5)) )]
  ""
  "ph_addhi3 %1 %2 %0
  \tph_addchi3 %6 %7 %5" )
```

Figure 24 Machine Description File showing how a complex *insn* sequence can be reduced down to two assembly instructions.

Compiler Output:

```
            ph_addhi3 r20 r16 r26
            ph_addchi3 r18 r14 r24
```

Figure 25 Assembly code produced by compiler.

Discussion: The compiler has correctly identified the *insn* code sequence and has substituted the two assembly instructions as expected. The `ph_` notation used for these instructions is to allow easy identification of those assembly instructions that were generated as a result of the peephole pass of the compiler. The use of `ph_addhi3` denotes an add *without* carry instruction. The `ph_addchi3` denotes an add *with* carry instruction.

Experiment 5

Aim: To attempt to translate one *insn* sequence into another *insn* sequence.

Conditions: The `addsi3` *insn* remains expanded in the machine description file. The previous `define_peephole` template is replaced with a `define_peephole2` template which allows an *insn* sequence to be matched and replaced with an alternate *insn* sequence.

```
(define_peephole2
 [(set (match_operand:HI 0 "general_operand" "")
                      (plus:HI (match_operand:HI 1 "general_operand" "")
                               (match_operand:HI 2 "general_operand" "")) )
  (set (cc0)
       (compare (match_dup 0) (match_dup 1)) )
  (set (match_operand:QI 3 "general_operand" "")
       (ltu:QI (cc0) (const_int 0)) )
  (set (match_operand:HI 4 "general_operand" "")
       (zero_extend:HI (match_dup 3)))
  (set (match_operand:HI 5 "general_operand" "")
       (plus:HI (match_operand:HI 6 "general_operand" "")
                (match_operand:HI 7 "general_operand" "")) )
  (set (match_dup 5)
       (plus:HI (match_dup 4)
                (match_dup 5)) ) ]
  ""
 [(set (match_dup 8)
       (plus:SI (match_dup 9)
                (match_dup 10)))]
 "{
  operands[8] = gen_rtx (GET_CODE(operands[5]), SImode, XEXP(operands[5],0));
  operands[9] = gen_rtx (GET_CODE(operands[6]), SImode, XEXP(operands[6],0));
  operands[10] = gen_rtx (GET_CODE(operands[7]), SImode, XEXP(operands[7],0));
  }")

;; Need this to match the above peephole2
(define_insn "alt_addsi3"
 [(set (match_operand:SI 0 "general_operand" "=g")
       (plus:SI (match_operand:SI 1 "general_operand" "g")
                (match_operand:SI 2 "general_operand" "g")))]
  ""
  "alt_addsi3 %1 %2 %0  ;(%1 plus %2)->%0" )
```

Figure 26 Machine Description File showing how a complex *insn* sequence can be translated into an alternate *insn* sequence.

Compiler Output:

```
            alt_addsi3 r18 r14 r24  ;(r18 plus r14)->r24
```

Figure 27 Assembly code produced by compiler.

Discussion: The compiler has correctly identified the *insn* code sequence and has substituted the *insn* sequence for the two assembly instructions as expected. The substitution sequence was later matched against the *alternate* `addsi3` *insn* during the assembly code generation phase of the compiler. The `alt_addsi3` instruction has been used to clearly identify that this instruction has come from the alternate `addsi3` *insn* and not the named *insn* pattern indicated in Figure 19 (which of course was removed from the machine description file after experiment 1).

Experiment 6

Aim: To test the compiler's ability to expand *insns* across multiple levels.

Conditions: The expansion for the `addsi3` *insn* remains. In addition to this, a `define_expand` template was created for the `addhi3` *insn*.

```
(define_expand "addhi3"
 [(set (match_operand:HI 0 "register_operand" "=r")
       (plus:HI (match_operand:HI 1 "register_operand" "g")
                (match_operand:HI 2 "register_operand" "g")))]
  ""
  "{
   operands[3] = gen_reg_rtx (QImode);

   emit_insn(gen_addqi3 (custom_subword(operands[0], 0, HImode),
                                        custom_subword(operands[1], 0, HImode),
                                        custom_subword(operands[2], 0, HImode) ));
   emit_insn(gen_cmpqi ( custom_subword(operands[0], 0, HImode),
                                        custom_subword(operands[1], 0, HImode) ));
   emit_insn(gen_sltu (operands[3]));
   emit_insn(gen_addqi3 (custom_subword(operands[0], 1, HImode),
                                        custom_subword(operands[1], 1, HImode),
                                        custom_subword(operands[2], 1, HImode) ));
   emit_insn(gen_addqi3 (custom_subword(operands[0], 1, HImode),
                                        operands[3],
                                        custom_subword(operands[0], 1, HImode) ));
   DONE;
   }" )
```

Figure 28 Machine Description File showing the expanded form of the *addhi3 insn*.

Compiler Output:

```
            addqi3 r21 r17 r27      ;(r21 plus r17)->r27
            cmpqi r27 r21     ;(r27-r21)->CC
            sltu r22
            addqi3 r20 r16 r26      ;(r20 plus r16)->r26
            addqi3 r22 r26 r26      ;(r22 plus r26)->r26
            cmphi r26 r20     ;(r26-r20)->CC
            sltu r22
            zero_extendqihi2 r22 r22      ;zero_extend(r22)->r22
            addqi3 r19 r15 r25      ;(r19 plus r15)->r25
            cmpqi r25 r19     ;(r25-r19)->CC
            sltu r30
            addqi3 r18 r14 r24      ;(r18 plus r14)->r24
            addqi3 r30 r24 r24      ;(r30 plus r24)->r24
            addqi3 r23 r25 r25      ;(r23 plus r25)->r25
            cmpqi r25 r23     ;(r25-r23)->CC
            sltu r18
            addqi3 r22 r24 r24      ;(r22 plus r24)->r24
            addqi3 r18 r24 r24      ;(r18 plus r24)->r24
```

Figure 29 Assembly code produced by compiler.

Discussion: This experiment proved the compiler was able to expand one *insn* which, in turn, lead to a second *insn* being expanded, i.e. the expansion of the `addsi3` *insn* lead to an expansion of the `addhi3` *insn*. The assembly code produced is almost identical to the code in Figure 21. The only difference is in the way that the carry information has been propagated. As a matter of interest, the code from this experiment was only 18 lines long whereas the code from Figure 21 consumes 21 lines of assembly.

The real significance of this experiment will become apparent in the next series of experiments as the focus moves towards reducing *insns* across multiple levels.

Experiment 7

Aim: To verify that a `peephole2` reduction can be used to reduce an *insn* sequence that has been created through multiple levels of expansions. In this case an attempt will be made to reduce the expanded form of `addhi3` back into a smaller *insn* sequence.

Conditions: Keeping the expanded form of `addsi3` and `addhi3`, a `define_peephole2` reduction was added to the machine description file.

```
 (define_peephole2
 [(set (match_operand:QI 0 "general_operand" "")
                      (plus:QI (match_operand:QI 1 "general_operand" "")
                               (match_operand:QI 2 "general_operand" "")) )
  (set (cc0)
       (compare (match_dup 0) (match_dup 1)) )
  (set (match_operand:QI 3 "general_operand" "")
       (ltu:QI (cc0) (const_int 0)) )
  (set (match_operand:QI 4 "general_operand" "")
       (plus:QI (match_operand:QI 5 "general_operand" "")
                (match_operand:QI 6 "general_operand" "")) )
  (set (match_dup 4)
       (plus:QI (match_dup 3)
                (match_dup 4)) ) ]
  ""
 [(set (match_dup 7)
       (plus:HI (match_dup 8)
                (match_dup 9)))]
 "{
  operands[7] = gen_rtx (GET_CODE(operands[4]), HImode, XEXP(operands[4],0));
  operands[8] = gen_rtx (GET_CODE(operands[5]), HImode, XEXP(operands[5],0));
  operands[9] = gen_rtx (GET_CODE(operands[6]), HImode, XEXP(operands[6],0));
  }")

;; Need this to match the above peephole2
(define_insn "alt_addhi3"
 [(set (match_operand:HI 0 "general_operand" "=g")
       (plus:HI (match_operand:HI 1 "general_operand" "g")
                (match_operand:HI 2 "general_operand" "g")))]
  ""
  "alt_addhi3 %1 %2 %0  ;(%1 plus %2)->%0" )
```

Figure 30 Machine Description File showing how the expanded `addhi3` *insn* sequence can be reduced again.

Compiler Output:

```
            alt_addhi3 r20 r16 r26  ;(r20 plus r16)->r26
            cmphi r26 r20     ;(r26-r20)->CC
            sltu r22
            zero_extendqihi2 r22 r22      ;zero_extend(r22)->r22
            alt_addhi3 r18 r14 r24  ;(r18 plus r14)->r24
            alt_addhi3 r22 r24 r24  ;(r22 plus r24)->r24
```

Figure 31 Assembly code produced by compiler.

Discussion: The existence of the `alt_addhi3` instructions in Figure 31 reveals that the compiler has correctly used the `define_peephole2` reduction.

Experiment 8

Aim: To verify that the `define_peephole2` reduction of the previous experiment can be further reduced using a `define_peephole` reduction.

Conditions: Whilst keeping the expanded form of `addsi3` and `addhi3` as well as the `define_peephole2` reduction of the previous experiment, an additional `define_peephole` reduction was added that attempted to recognize the *insn* sequence created by the `define_peephole2` reduction of Experiment 7. The `alt_addhi3` *insn* from Experiment 7 was left in the machine description file also.

```
(define_peephole
 [(set (match_operand:HI 0 "general_operand" "")
                       (plus:HI (match_operand:HI 1 "general_operand" "")
                                (match_operand:HI 2 "general_operand" "")) )
  (set (cc0)
       (compare (match_dup 0) (match_dup 1)) )
  (set (match_operand:QI 3 "general_operand" "")
       (ltu:QI (cc0) (const_int 0)) )
  (set (match_operand:HI 4 "general_operand" "")
       (zero_extend:HI (match_dup 3)))
  (set (match_operand:HI 5 "general_operand" "")
       (plus:HI (match_operand:HI 6 "general_operand" "")
                (match_operand:HI 7 "general_operand" "")) )
  (set (match_dup 5)
       (plus:HI (match_dup 4)
                (match_dup 5)) )]
  ""
  "ph_addhi3 %1 %2 %0
  \tph_addchi3 %6 %7 %5" )
```

Figure 32 Machine Description File showing how the expanded `addhi3` *insn* sequence can be reduced again.

Compiler Output:

```
            ph_addhi3 r20 r16 r26
            ph_addchi3 r18 r14 r24
```

Figure 33 Assembly code produced by compiler.

Discussion: The reader will note that the resulting assembly code from this experiment is identical to that of experiment 4 however the path via which it was arrived at by the compiler is different. In the case of experiment 4, the assembly code was produced directly from the expanded `addsi3` *insn* sequence. In this case, two levels of expansion were performed (`addsi3` & `addhi3`). The expanded `addhi3` *insn* sequence was then reduced by a `define_peephole2` template which in turn was reduced by a `define_peephole` template that returned the assembly code listing in Figure 33.

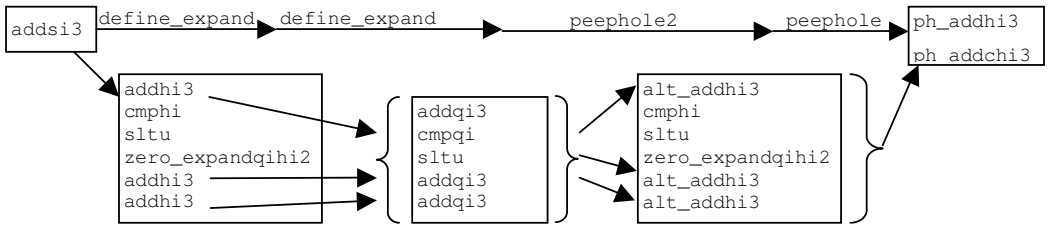


Figure 34 Visualization of the successive expansion and reduction processes that lead to the assembly code of Figure 33.

Experiment 9

Aim: To determine whether the compiler is able to perform `define_peephole2` reductions on *insn* sequences that have themselves been created as a result of a previous `define_peephole2` reduction.

Conditions: The machine description file was the same as in the previous experiment however the `define_peephole` reduction indicated in Figure 32 was replaced with the `define_peephole2` of Figure 26.

Compiler Output:

```
alt_addhi3 r20 r16 r26  ;(r20 plus r16)->r26
cmphi r26 r20     ;(r26-r20)->CC
sltu r22
zero_extendqihi2 r22 r22      ;zero_extend(r22)->r22
alt_addhi3 r18 r14 r24  ;(r18 plus r14)->r24
alt_addhi3 r22 r24 r24  ;(r22 plus r24)->r24
```

Figure 35 Assembly code produced by compiler.

Discussion: It appears that the second peephole2 optimization pass has not occurred since the assembly code produced is the same as Figure 31. Had the second peephole2 reduction occurred then the assembly code of Figure 37 would be expected.

Further investigation revealed that the order of the `define_peephole2` definitions in the machine description file did not impact on the result.

Experiment 10

Aim: To determine whether the compiler is able to perform `define_peephole2` reductions on *insn* sequences that have themselves been created as a result of a previous `define_peephole2` reduction.

Conditions: The machine description file was the same as in the previous experiment. The compiler source code was modified so that the peephole2 optimisation code was run a second time.

```
if (optimize > 0 && flag_peephole2)
  {
    timevar_push (TV_PEEPHOLE2);
    open_dump_file (DFI_peephole2, decl);

    peephole2_optimize (rtl_dump_file);
    /* MAH 18/8/05 add a second pass of the peephole optimizer */
    peephole2_optimize(rtl_dump_file);

    close_dump_file (DFI_peephole2, print_rtl_with_bb, insns);
    timevar_pop (TV_PEEPHOLE2);

  }
```

Figure 36 Modified version of `Toplev.c` taken from the main GCC source code.

Compiler Output:

```
        alt_addsi3 r18 r14 r24  ;(r18 plus r14)->r24
```

Figure 37 Assembly code produced by compiler.

Discussion: The compiler modification worked in that successive `define_peephole2` reductions have executed and reduced the assembly code output to a single instruction.

Experiment 11

Aim: To determine whether *insn* reductions can be performed on sequences that extend across basic blocks.

Conditions: Two forms of an open-coded addition routine are compiled and the associated compiler output observed. The first open-coded 16-bit addition avoided any branching code and was therefore contained within a single basic block.

```
#define WORDS_BIG_ENDIAN

typedef unsigned int QItype __attribute__ ((mode (QI)));
typedef unsigned int HItype __attribute__ ((mode (HI)));

typedef union {
#ifdef WORDS_BIG_ENDIAN
  struct {QItype hi; QItype lo;} s;
#else
  struct {QItype lo; QItype hi;} s;
#endif
  HItype wd;
} HIunion;

HItype oc_addhi3 (HItype A, HItype B)
{
  HIunion Z;
  Z.s.lo = ((HIunion)A).s.lo + ((HIunion)B).s.lo;
  Z.s.hi = (Z.s.lo < ((HIunion)A).s.lo);
  Z.s.hi = Z.s.hi + ((HIunion)A).s.hi;
  Z.s.hi = Z.s.hi + ((HIunion)B).s.hi;
  return Z.wd;
}
```

Figure 38 Example of an open-coded 16-bit addition function contained within a single basic block.

The second open-coded 16-bit addition made use of an if/then/else block for carry propagation.

```
#define WORDS_BIG_ENDIAN

typedef unsigned int QItype __attribute__ ((mode (QI)));
typedef unsigned int HItype __attribute__ ((mode (HI)));

typedef union {
#ifdef WORDS_BIG_ENDIAN
  struct {QItype hi; QItype lo;} s;
#else
  struct {QItype lo; QItype hi;} s;
#endif
  HItype wd;
} HIunion;

HItype oc2_addhi3 (HItype A, HItype B)
{
  HIunion Z;
  Z.s.lo = ((HIunion)A).s.lo + ((HIunion)B).s.lo;
  if (Z.s.lo < ((HIunion)A).s.lo)
    Z.s.hi = ((HIunion)A).s.hi + ((HIunion)B).s.hi + 1;
  else
    Z.s.hi = ((HIunion)A).s.hi + ((HIunion)B).s.hi;
  return Z.wd;
}
```

Figure 39 Example of an open-coded 16-bit addition function that extends across multiple basic blocks.

Compiler Output:

```
oc_addhi3:
/* prologue: frame size=0 */
/* prologue end (size=0) */
            movhi r22 r20     ;r22->r20
            addqi3 r25 r21 r19      ;(r25 plus r21)->r19
            cmpqi r19 r25     ;(r19-r25)->CC
            sltu r18
            addqi3 r18 r24 r18      ;(r18 plus r24)->r18
            addqi3 r18 r22 r18      ;(r18 plus r22)->r18
            zero_extendhisi2 r18 r24      ;zero_extend(r18)->r24
            movsi r24 r22     ;r24->r22
/* epilogue: frame size=0 */
            return
/* epilogue end (size=1) */
/* function oc_addhi3 size 17 (16) */
            .size    oc_addhi3, .-oc_addhi3
.global     oc2_addhi3
            .type    oc2_addhi3, @function
```

Figure 40 Assembly code resulting from source code of Figure 38

```
.global     oc2_addhi3
            .type    oc2_addhi3, @function
oc2_addhi3:
/* prologue: frame size=0 */
/* prologue end (size=0) */
            addqi3 r25 r23 r19      ;(r25 plus r23)->r19
            cmpqi r19 r25     ;(r19-r25)->CC
            bgeu .L3
            addqi3 r24 r22 r24      ;(r24 plus r22)->r24
            addqi3 r24 #1 r18 ;(r24 plus #1)->r18
            jump .L4
.L3:
            addqi3 r24 r22 r18      ;(r24 plus r22)->r18
.L4:
            zero_extendhisi2 r18 r24      ;zero_extend(r18)->r24
            movsi r24 r22     ;r24->r22
/* epilogue: frame size=0 */
            return
/* epilogue end (size=1) */
/* function oc2_addhi3 size 19 (18) */
            .size    oc2_addhi3, .-oc2_addhi3
```

Figure 41 Assembly code resulting from source code of Figure 39

Discussion: The structure of the peephole optimizer is such that it only searches for *insn* substitution opportunities within a single basic block. It is therefore not possible for *insn* groups which span multiple basic blocks to be reduced.

GCC Structural Limitation Findings

Based on the experiments carried out as part of this study, the following conclusions were made.

Reduction Optimizations are not iterative.

GCC has been coded to perform the expansion and reduction phases only once and in a predefined order. The specific passes of the compiler and the order in which they run is provided in [135] and is controlled from the top-level GCC source code file which is aptly

named `toplev.c`. There is no mechanism to force optimization passes to rerun or run out of order other than by changing the source code of the compiler.

The peephole2 optimization pass is designed to transform *insn* sequences of one form into another form. Unfortunately because it is only run once, *insn* sequences created as a result of an initial pass of the peephole2 optimizer will not be further reduced using this *insn* reduction mechanism. Experiment 9 demonstrated that it is possible to modify the compiler to make successive peephole2 optimization passes however this then poses the question as to the appropriate number that successive passes should be made.
A better implementation would be for the peephole2 optimizer to keep a record of whether or not any *insn* substitutions had been made during the previous pass. If any had been made then it is possible that the new *insn* list could be further reduced and therefore an additional peephole2 optimization pass would be warranted. Conversely, if no substitutions have been made then no further optimization passes would be necessary. It is possible that employing an iterative optimization strategy such as this may lead to an infinite loop of *insn* transformations and so an upper bound on optimization passes would be necessary to prevent such a situation occurring.

The peephole optimization pass translates an *insn* sequence to assembly code and is run as part of the final pass of the compiler. Since there is no compiler pass directed at translating assembly code sequences from one form to another, it does not make any sense to force this pass of the compiler to run more than once. Because the peephole optimizer runs *after* the peephole2 optimization pass, the peephole optimizer can be used to perform a second level of optimization however developing templates to cover all possibilities requires extensive work and can lead to increases in compilation time. If optimization capabilities are to be improved then it is in the peephole2 pass that effort should be concentrated and not in the peephole pass.

Peephole optimizations can not occur across basic blocks.
The structure of the peephole optimizer is such that it only searches for *insn* substitution opportunities within a single basic block. This is a significant limitation with implications that flow beyond open-coded math routines. Given a choice, *insn* expansion routines should limit the use of branching code. This will ensure the greatest opportunity for *insn* optimizations to be made during the *insn* reduction phases of the compiler.

Summary and Concluding Remarks

This chapter has outlined both the positive and negative aspects of GCC that effect its suitability for Compiler Directed Codesign. Where limitations have been identified, they have been characterized and suggested means for minimizing their effects have been proposed. In particular, it was noted that the GCC compiler requires some modifications in order to be able to perform iterative *insn* reductions. The reduction of i*nsns* across basic blocks is currently not possible and would require extensive modifications to GCC in order for this to be supported.

The structural aspects of GCC that were presented in this chapter demonstrate GCC's ability to both expand and contract instruction sequences. This is a core requirement for Compiler Directed Codesign as it allows for complex instructions to be expanded into a sequence of less complex instructions and vice versa. This means that where the necessary silicon resources are available, complex hardware units can be included in the processor and invoked using a single instruction. In the situation where silicon resources are limited, complex instructions can be broken down into a sequence of simpler instructions.

The final challenge in using GCC for Compiler Directed Codesign is in the arena of retargetability. It is currently estimated that retargeting GCC to a new processor takes about 3-4 man-months for someone who knows what they are doing [109]. This timeframe is clearly unacceptable for an iterative design process and so the next chapter will explain the specific structure of the Compiler Directed Codesign apparatus and the modifications made to GCC to facilitate its *rapid* retargeting.

Chapter 5.

BUILDING THE COMPILER DIRECTED CODESIGN PLATFORM

The previous two chapters demonstrated how GCC's ability to expand or reduce instruction sequences makes it well suited to the task of Compiler Directed Codesign. At this point, the most critical problem to be overcome is that of rapidly retargeting GCC for a new processor and exposing the performance impact that including or excluding various instructions will have on a given application.

In this chapter I will explain the modifications and supporting programs that were added to the GCC environment to better facilitate the codesign process and to speed up the targeting process.

Exposing GCC internals

GCC's compilation process was presented in detail in Chapter 4 but is repeated again here for reference.

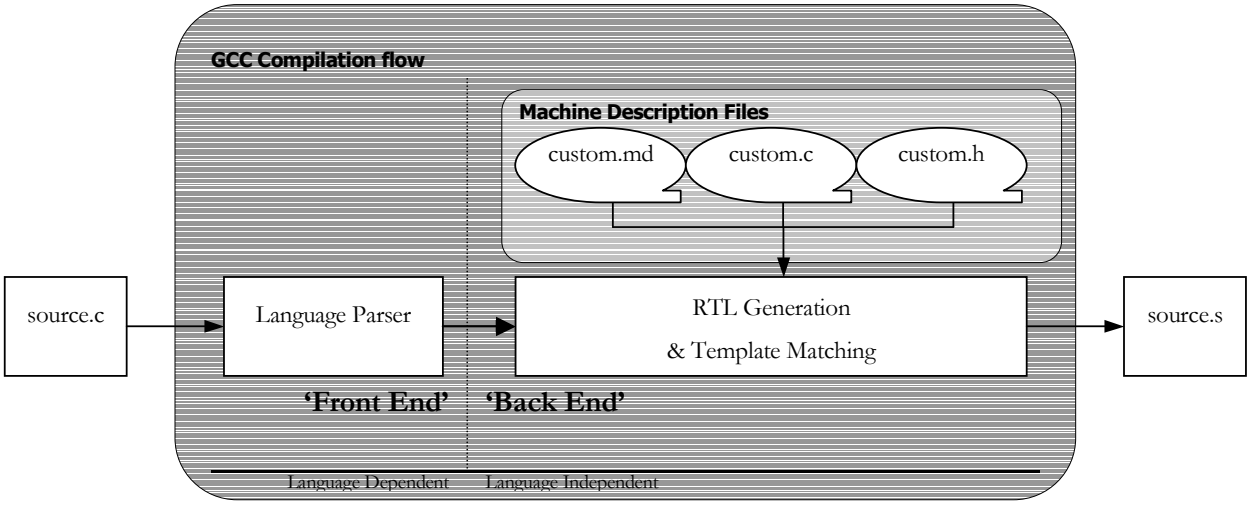


Figure 42 Simplified View of the GCC Compiler flow

Compilation begins with the front end of the compiler reading the application source code and creating a parse tree. This is then passed to the back end of the compiler for conversion

to an intermediate RTL representation (called an *insn list*) before being mapped to the instruction set of the target processor.

The RTL *insn* list is created using RTL templates found in the target processor's machine description file (`custom.md`). There are a total of 118 primitive RTL commands that have been hard-coded into GCC and each has a unique name that identifies its operation. In addition to this, GCC also specifies several data *modes* that are used to identify the width of any operand data. By combining the primitive RTL commands with the various data modes, a collection of several thousand unique RTL instruction patterns are possible.

At the time when the *insn* list is first generated, GCC searches the machine description file for specific RTL templates according to their name and data mode. If an exact match can not be found, the compiler will resort to an alternate strategy that uses different RTL templates. If all alternate strategies are exhausted, the compiler will abort with an error message.

Because the RTL primitives defined by GCC represent basic computational operations, they can be used as an indicator to GCC of the hardware resources that are available on the target processor. For instance, a processor that is capable of performing 16-bit additions would include in its machine description file a standardized `addhi3` RTL template that would be used to create a corresponding `addhi3` *insn* in the *insn* list whenever the application code called for a 16-bit addition. If the `addhi3` *insn* was not made available, GCC would be forced to find an alternate strategy such as using several 8-bit addition *insns* along with some carry propagation *insns*.

Once a complete *insn* list has been constructed, GCC will optionally perform a number of translations and modifications to the *insn* list in an attempt to optimise it. At the conclusion of the optimisation process, GCC will then traverse the resulting *insn* list and attempt to match each *insn* with an RTL template in the machine description file. Where a match is found, the assembly code fragment that is part of the matching RTL template will be exported to produce the final assembly code listing and conclude the compilation process.

Most real world processors will not have a 1:1 correspondence between the number of *insns* in the *insn* list and the total number of exported assembly statements. For example some *insns* will be combined together and represented by a single assembly statement whereas others will be represented by multiple assembly statements. In general, however, the *relative* size of the *insn* list and the *relative* size of the exported assembly code listing will correlate.

Compiler Directed Codesign exploits this correlation to provide a relative measure of the impact on performance that including or excluding various hardware resources in a target

processor will have. For example, when all possible RTL template permutations are listed in the machine description file, the compiler will always implement the application code using the least number of *insns*. This represents a best case performance scenario but would require a processor with a relatively high amount of hardware resources. By removing RTL templates from the machine description file, a processor that consumes less hardware resources can be described to GCC. When the application is recompiled, the compiler will be forced to find alternate strategies that will, in general, use more *insns* leading to a slower execution speed.

By iteratively removing RTL templates and measuring the impact it has on the total *insn* count, a curve can be plotted that tracks the relationship between hardware resource and relative execution speed. From this curve, the system designer can then choose the best point at which performance requirements are met with minimal hardware resources.

The key to the success of the Compiler Directed Codesign approach is therefore based on the ability to quickly and easily update the list of available RTL templates in the machine description file as well as the ability to accurately observe the total number of *insns* used for each compilation of the target application. Following is a discussion of the various additions and modifications made to GCC to fulfil these requirements.

Automating the Machine Description File Generation Process

The machine description file (`custom.md`) is the point in which all *insn* templates and assembly translations are made for a given processor. Correctly defining this file from scratch is an arduous task and can take several months for a new architecture. The goal was to dramatically decrease the time required to create and modify this file.

Because all named *insns* are designed to perform a known function and the form of the template that defines them can be standardized to some extent, there was an opportunity to automate the template generation. In practice a *real-world* target architecture may have limitations that would require further refinement of the *insn* templates. However, the goal here was to use the RTL representation as a starting point for a processor description and so the RTL templates were given a high degree of flexibility.

In order to speed up the generation of the machine description file for a custom processor (custom.md), a sequence of programs were developed. Complete code listings can be found in the electronic appendix. Brief explanations of each of the programs are presented below:

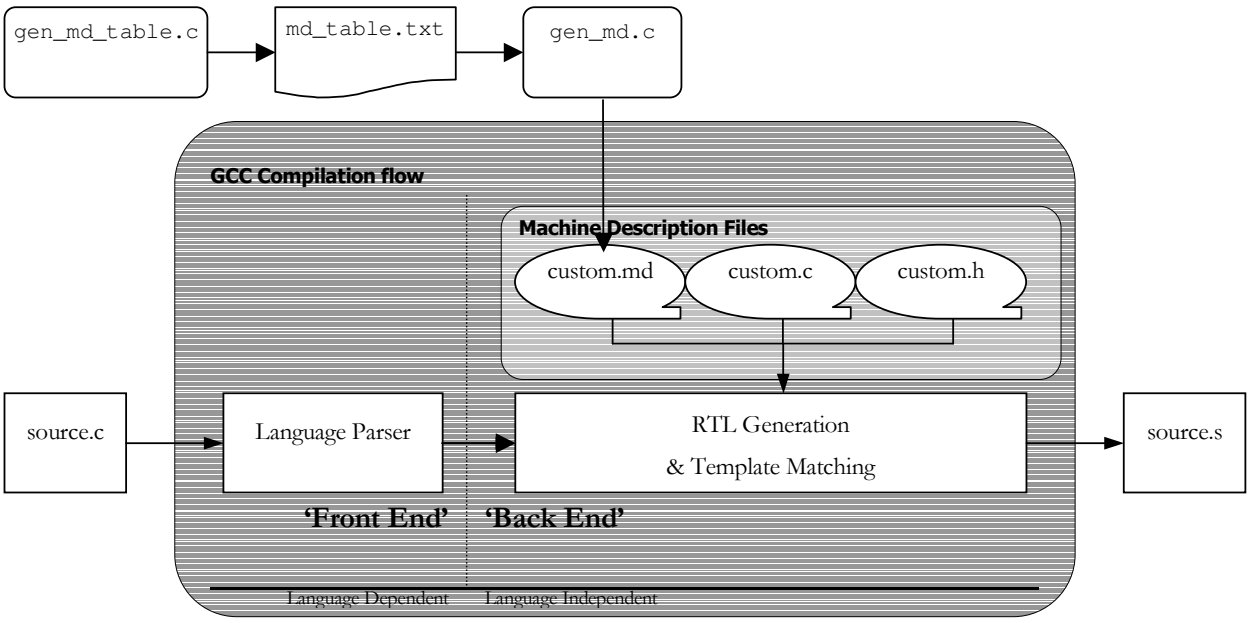


Figure 43 Automating the generation of the Machine Description file.

gen_md_table.c

This program builds an array called `insn_modes[]` which contains all of the *insn*/mode combinations possible in GCC and indicates which standard *insns* are to be defined for each of the different machine modes. Each element within the array can have a value of `UNCONSTRAINED(1)`, `DISABLED(-1)`, or `DEFAULT(0)` and this indicates whether the mode is explicitly enabled, explicitly disabled, or assumes a default state.

Each of the GCC modes are further classified according to a mode *class* that groups similar modes together. For example all integer modes are grouped into a class called `MODE_INT`; all floating point modes are grouped into another class called `MODE_FLOAT`. Grouping machine modes into classes allows entire classes of *insn* modes to be readily enabled or disabled. Mode class information is held in an array called `mode_group_state[]`.

A hierarchy exists in the way that *insn*/mode combinations are enabled or disabled and is given below in order of highest to lowest priority:

1. Individual *insns* can be enabled/disabled for a specific mode by setting an appropriate value in the `insn_modes[]` array. For example, to disable the `cmpm` (compare) *insn* for the `HImode` (16-bit) mode;

```
insn_modes[INSNcmpm][HImode] = DISABLED;
```

2. All *insns* of a given mode can be enabled/disabled by setting an appropriate value in the `MAX_INSN` element of the `insn_modes[]` array. For example, to disable all `SImode` (32-bit) *insns*;

```
insn_modes[MAX_INSN][SImode] = DISABLED;
```

3. An *insn* can be enabled/disabled across all modes by setting an appropriate value in the `MAX_MACHINE_MODE` element of the `insn_modes[]` array. For example, to disable the `mulm3` (multiply) *insn* across all modes;

```
insn_modes[INSNmulm3][MAX_MACHINE_MODE] = DISABLED;
```

4. Entire mode classes can be enabled/disabled by setting an appropriate value in the `mode_group_state[]` array. For example, to disable all floating point modes;

```
mode_group_state[MODE_FLOAT] = DISABLED;
```

To create a new machine description file, the developer would edit the `set_user_constraints()` function in the `gen_md_table.c` file to enable/disable specific *insn*/mode combinations. `gen_md_table.c` would then be compiled and run to produce a configuration text file that can then be used by the `gen_md` program to build the final machine description file.

`gen_md.c`

The `gen_md` program reads the entire contents of the text file produced by `gen_md_table` and emits a new machine description file. *Insn* templates will be emitted for a given mode if they satisfy the following rules:

If an explicit enable/disable value is set for the *insn*/mode combination in question then emit/suppress the *insn* template;
Else
  If an explicit enable/disable value is set for the mode in question then emit/suppress the *insn* template;
  Else
    If an explicit enable/disable value is set for the *insn* in question then emit/suppress the *insn* template;

Else

If an explicit enable/disable value is set for the class of the mode in question then emit/suppress the *insn* template;

Else

If the width of the mode is greater than the default width of the processor then suppress the *insn* template;

Else

Emit the *insn* template.

The machine description file generated by `gen_md.c` must then be compiled into GCC before the target application is compiled again.

A long term goal of this research (beyond the timeframe of this thesis) is to build a system that is capable of determining an optimal processor description with minimal user interaction. The process of creating a machine description file was therefore broken into two phases as this will lend itself better to a fully automated process in the future.

Extracting *Insn* Usage Information from the Compiler.

The key idea of Compiler Directed Codesign is to use information extracted from the compiler to direct processor development for a given application. As previously discussed, this involves making iterative changes to the machine description file to indicate the availability or lack of specific hardware resources, and then measuring the impact that this has on the *insn* list produced when the target application is compiled.

The two programs discussed above help to simplify the process of updating the machine description file but the real challenge exists at the point of tracking *insn* usage. Without this information the impact of changes made to the machine description file can not be quantified and there is no guidance as to which RTL template should be enabled or disabled in the next iteration.

The complexity of GCC is such that it would seem that any minor modification to the compiler core represents a major work and inserting the functionality required to track *insn* usage was much more than a minor modification. A number of modification strategies were attempted before the final solution became apparent.

Initial Strategy

The containing type for a GCC *insn* is a structure called an `rtx` (register transfer language expression). The set of GCC source files that define the `rtx` are `rtl.h`, `rtl.c` and `rtl.def`. In the first attempt to track *insn* usage within the compiler, a new field (called `creator_id`) was added to the rtx structure that would allow the original RTL template that was used in the *insn*'s creation to be identified.

The idea was that this field would be carried with the `rtx` through all of the compiler passes and could be extracted into a text file as part of the rtl debugging dump[4]. An additional macro was also defined called `INSN_CREATOR()` that returned the `creator_id` of a given `rtx`.

GCC has a number of ancillary programs that read the machine description file and produce various modules that are compiled into the final compiler. One such program is `genemit.c`. This program is responsible for taking *insns* defined in the machine description file and producing a C file called `insn-emit.c` with functions relating to each *insn* that can be called by the compiler during compilation. For example, the *insn* template defined in the machine description file as:

```
(define_insn "movqi"
 [(set (match_operand:QI 0 "general_operand" "=g")
       (match_operand:QI 1 "general_operand" "g"))]
  ""
  "movqi %1 %0       ;%1->%0" )
```

Figure 44 Example of a `movqi` *insn* template taken from the machine description file.

would be converted by `genemit.c` into a function contained within `insn-emit.c` called `gen_movqi()`:

```
rtx
gen_movqi (operand0, operand1)
     rtx operand0;
     rtx operand1;
{
  return (gen_rtx_SET (VOIDmode,
    operand0,
    operand1));
}
```

Figure 45 Original `gen_movqi` function generated by `genemit.c` from the *insn* template defined in Figure 44.

In order to keep *insn* creation information, `genemit.c` was modified to produce a table that linked all of the *insn* IDs with their names as well as storing *insn* IDs within the rtx structure. An example of the new code produced in `insn-emit.c` is given below (Figure 46).

[4] An RTL debugging dump can be made by using the compiler switch `-dr` when compiling an application.

```
rtx
gen_movqi (operand0, operand1)
     rtx operand0;
     rtx operand1;
{
  rtx _me;
  creator = 2;
  _me = gen_rtx_SET (VOIDmode,
            operand0,
            operand1);
  INSN_CREATOR(_me) = 2;
  return _me;
}
```

Figure 46 Modified `gen_movqi` function to include *insn* creation information.

Exposing the *insn* Usage Information

Once the *insn* creation ID had been successfully added to the `rtx` structure and inserted into it at the point it was created, the final task of extracting *insn* usage information after application compilation was all that remained.

GCC contains a function called `print-rtl()` that takes an `rtx` and prints its structure, in a human readable format, to a file. The routine `close_dump_file()` walks through the entire `rtx` tree (*insn* list) and successively calls `print-rtl()` for each `rtx` that it encounters. By creating a new function called `print_insn_usage()` and substituting it for `print-rtl()`, it was possible to override the rtl dump process. `print_insn_usage()` simply extracted the creation ID and machine mode of each `rtx` passed to it and printed them out to a log file.

An example of the output produced by `print_insn_usage()` is given below (Figure 47).

```
addhi3 called using mode HI
movhi called using mode HI
movhi called using mode HI
movhi called using mode HI
tsthi called using mode HI
bgt called using mode VOID
jump called using mode VOID
movhi called using mode HI
addhi3 called using mode HI
```

Figure 47 Extract of rtl dump file generated by `print_insn_usage()`.

`print_insn_usage()` emits a line of information for each `rtx` passed to it. A simple script file was created to summarize the *insn* usage information and display it in a more palatable form. An example of the summary information produced is provided in Figure 48.

```
HI              ashlhi3 has been called     1 times ( 0.2320%)
HI               neghi2 has been called     1 times ( 0.2320%)
HI     zero_extendqihi2 has been called     2 times ( 0.4640%)
HI               *dummy has been called     3 times ( 0.6961%)
HI          extendqihi2 has been called     3 times ( 0.6961%)
HI               iorhi3 has been called     3 times ( 0.6961%)
HI                 call has been called     4 times ( 0.9281%)
HI               subhi3 has been called     5 times ( 1.1601%)
HI              ashrhi3 has been called     7 times ( 1.6241%)
HI                cmphi has been called     7 times ( 1.6241%)
HI               andhi3 has been called     8 times ( 1.8561%)
HI                tsthi has been called    15 times ( 3.4803%)
HI               addhi3 has been called    40 times ( 9.2807%)
HI                movhi has been called   261 times (60.5568%)
QI               andqi3 has been called     1 times ( 0.2320%)
QI                  seq has been called     2 times ( 0.4640%)
QI                tstqi has been called     2 times ( 0.4640%)
QI               iorqi3 has been called     3 times ( 0.6961%)
QI                movqi has been called    15 times ( 3.4803%)
VOID                bne has been called     1 times ( 0.2320%)
VOID                bge has been called     3 times ( 0.6961%)
VOID                bgt has been called     3 times ( 0.6961%)
VOID                ble has been called     3 times ( 0.6961%)
VOID                blt has been called     4 times ( 0.9281%)
VOID          call_value has been called     5 times ( 1.1601%)
VOID                beq has been called     8 times ( 1.8561%)
VOID               jump has been called    21 times ( 4.8724%)
================================================
27 unique insns used across 431 total insns called.
```

Figure 48 Summary information detailing *insn* usage.

According to the last line of Figure 48, 27 different *insns* were required to implement the program. The summary information indicates that operand data used the modes QI (quarter integer or 8-bit) and HI (half integer or 16-bit). Of these *insns*, the `movhi` *insn* accounted for 60% of all *insns* used. This analysis is based on a static view of the program – i.e. not running.

An Improved Method of Extracting *insn* Usage Information.

The *insn* extraction mechanism described in the previous section was based on the assumption that compilation would be aborted just prior to assembly code generation. Subsequently the focus was on extracting *insn* information whilst the *insn* structures were still 'live' within the compiler (i.e. during compilation). There are a number of deficiencies with this approach as it can only be used as a static analysis tool and so an alternate approach was developed that would be able to track dynamic usage.

By allowing compilation to continue to the point of assembly creation, the actual assembly code could be used to measure *insn* usage information. The key to this technique is in the way that the RTL templates are defined in the machine description file. Essentially there are two sections to an RTL template. The first section defines the actual RTL structure used when the *insn* list is first created, and the second section defines an assembly fragment that will be emitted when the *insn* list is converted into assembly code.

By ensuring that each named *insn* within the machine description file was programmed to produce an assembly opcode with the same name as the RTL template, *insn* usage information could be extracted directly from the assembly code output.

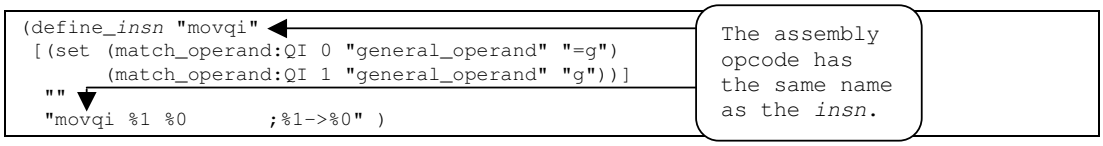


Figure 49 Example of a movqi *insn* template with matching opcode and *insn* name.

This enhanced method offers a number of key advantages:

1. **Complicated modifications of the compiler internals are not necessary.**

   This is particularly important given that GCC is developed by many developers for many purposes. Modifications made to the compiler internals will get lost in subsequent GCC releases unless application is made to the GCC committee to have the modifications applied to the GCC source code. Any modifications to the GCC code base must be assessed in the context of how they benefit *all* GCC users. As the code modifications proposed in this thesis are for largely experimental purposes, there is no guarantee that they would be accepted into the permanent code base.

2. **Peephole optimizations can be included.**

   The peephole optimization pass is part of the assembly code generation pass of the compiler. The initial *insn* tracking method counted *insns* prior to the completion of the assembly code generation pass and therefore would not benefit from peephole optimization opportunities.

3. **Assembly code is required for a complete solution**

   Given that the end goal of this research is to specify a processor model that would service a specific application, an assembly code listing will eventually be required if the code is to run on an actual processor. This approach allows for this.

4. **Function Prologue/Epilogue code can be included in the *insn* count**

   The compiler does not create an *insn* list for prologue and epilogue code. This code is simply inserted directly as assembly instructions as part of the assembly code generation pass. The previous *insn* accounting method would miss all function epilogue/prologue information and would therefore underestimate the true *insn* usage values.

5. **Assembly code information can be used for dynamic analysis**

This will be discussed further in the next section but it is important to state that the previous *insn* accounting method is unable to make use of profiling information and is therefore limited to static analysis only. By effectively including *insn* information in the assembly listing it was possible to track the dynamic usage of *insns*.

Static versus Dynamic Analysis

By tallying up the *insns* used to implement a given application, the designer can get a sense of the most frequently used *insns* and the width of data that they operate on. But a static analysis on its own is inadequate. It provides *insn* frequency information on the assumption that every line of code in the application is executed exactly once, which is rarely occurs in practice. Most applications will contain loops and branches that will lead to some lines being executed many times whilst other lines may never be executed at all.

Obtaining dynamic *insn* usage information in a codesign environment is quite difficult as it generally requires an execution platform to be present or at least some means to simulate it. Given that the Compiler Directed Codesign approach involved making successive iterations of the processor model, it was desirable to devise a way to obtain dynamic *insn* usage information without the need for advanced execution or simulation models being required.

Using `gcov` to extract dynamic *insn* usage metrics

A tool that is often used as a companion with GCC to test for code coverage is `gcov`. To use it, an application must first be compiled by GCC in the following manner:

```
gcc -fprofile-arcs -ftest-coverage [additional_options] sourcefile.c
```

Figure 50 Compiling an application for use with `gcov`.

This tells the compiler to insert additional code into the application's object files so that profiling information can be recorded. Two files used for reconstructing the program flow will also be created (`sourcefile.bb` and `sourcefile.bbg`). The `.bb` file contains a list of source files (including headers), functions within those files, and line numbers corresponding to each basic block[5] of code. The `.bbg` file contains a list of program flow arcs (possible branches taken from one basic block to another) for each function.

When a program that has been compiled with the options listed above is run, a file containing a count of how many times each arc in the program is executed is dynamically

[5] A basic block is a section of code containing no loops or branches

created and given the extension `.da`. A code coverage analysis can then be done by invoking `gcov` with the name of the original source file as its argument:

```
gcov [options] sourcefile.c
```

Figure 51 Invoking `gcov`.

`Gcov` will then create a logfile with the extension `.gcov` that lists the original source code annotated with line numbers and the number of times each line of code was executed:

```
      256:  239:        vpdiff = step >> 3;
      256:  240:        if (delta & 4)
       66:  241:            vpdiff += step;
      256:  242:        if (delta & 2)
      102:  243:            vpdiff += step >> 1;
      256:  244:        if (delta & 1)
      149:  245:            vpdiff += step >> 2;
```

Figure 52 Extract taken from a `.gcov` file.

Reading from the left, the first column is the execution count for that line. If the line is not an executable line it will be annotated with a '−'. If it is executable but its execution count is 0, it will be annotated with a '#####'. The second column contains the line number of the original source code and the final column contains the actual source code.

In most situations where GCC is used in a non-codesign capacity, it will be expected to perform the compilation process as well as invoking the downstream assembler and linker to produce an executable program. In the case of Compiler Directed Codesign, however, no assembler or linker exists because the processor is yet to be defined. Fortunately, GCC offers a compiler option that aborts the build process once the assembly file has been created. Furthermore, additional options allow for source line information to be included in the assembly listing so that it can be correlated back to the original source code. By matching the line numbers provided in the assembly listing against the line numbers and execution counts provided in the .gcov file, it is possible to determine dynamic *insn* information even without a simulator or execution platform other than a basic PC.

```
240:/ADPCM_Verified.c ****     if (delta & 4)
1395                    .stabn 68,0,240,.LM74-adpcm_decoder
1396            .LM74:
1397                    andhi3 (r28+#15) #4 r24 ;((r28+#15) and #4)->r24
1398                    tsthi r24       ;tst(r24)->CC
1399                    beq .L63
241:/ADPCM_Verified.c ****     vpdiff += step;
1400                    .stabn 68,0,241,.LM75-adpcm_decoder
1401            .LM75:
1402 ???? 0000          movqi (r28+#25) (r28+#64)       ;(r28+#25)->(r28+#64)
1403 ???? 0000          movqi (r28+#26) (r28+#65)       ;(r28+#26)->(r28+#65)
1404 ???? 0000          movqi (r28+#27) (r28+#66)       ;(r28+#27)->(r28+#66)
1405 ???? 0000          movqi (r28+#28) (r28+#67)       ;(r28+#28)->(r28+#67)
1406 ???? 0000          movqi (r28+#17) (r28+#68)       ;(r28+#17)->(r28+#68)
1407 ???? 0000          movqi (r28+#18) (r28+#69)       ;(r28+#18)->(r28+#69)
1408 ???? 0000          movqi (r28+#19) (r28+#70)       ;(r28+#19)->(r28+#70)
1409 ???? 0000          movqi (r28+#20) (r28+#71)       ;(r28+#20)->(r28+#71)
1410                    addqi3 (r28+#67) (r28+#71) (r28+#75)    ;((r28+#67) plus (r28+#71))->(r28+#75)
1411 ???? 0000          movqi #1 (r28+#76)      ;#1->(r28+#76)
1412                    cmpqi (r28+#75) (r28+#67)       ;((r28+#75)-(r28+#67))->CC
1413                    bltu .L64
1414 ???? 0000          movqi #0 (r28+#76)      ;#0->(r28+#76)
1415            .L64:
1416                    addqi3 (r28+#66) (r28+#70) (r28+#74)    ;((r28+#66) plus (r28+#70))->(r28+#74)
1417 ???? 0000          movqi #1 (r28+#77)      ;#1->(r28+#77)
1418                    cmpqi (r28+#74) (r28+#66)       ;((r28+#74)-(r28+#66))->CC
1419                    bltu .L65
1420 ???? 0000          movqi #0 (r28+#77)      ;#0->(r28+#77)
1421            .L65:
1422                    addqi3 (r28+#76) (r28+#74) (r28+#78)    ;((r28+#76) plus (r28+#74))->(r28+#78)
1423 ???? 0000          movqi #1 (r28+#79)      ;#1->(r28+#79)
1424                    cmpqi (r28+#78) (r28+#74)       ;((r28+#78)-(r28+#74))->CC
1425                    bltu .L66
1426 ???? 0000          movqi #0 (r28+#79)      ;#0->(r28+#79)
1427            .L66:
1428                    iorqi3 (r28+#77) (r28+#79) (r28+#77)    ;((r28+#77) ior (r28+#79))->(r28+#77)
1429 ???? 0000          movqi (r28+#78) (r28+#74)       ;(r28+#78)->(r28+#74)
1430                    addqi3 (r28+#65) (r28+#69) (r28+#73)    ;((r28+#65) plus (r28+#69))->(r28+#73)
1431 ???? 0000          movqi #1 (r28+#80)      ;#1->(r28+#80)
1432                    cmpqi (r28+#73) (r28+#65)       ;((r28+#73)-(r28+#65))->CC
1433                    bltu .L67
1434 ???? 0000          movqi #0 (r28+#80)      ;#0->(r28+#80)
1435            .L67:
1436                    addqi3 (r28+#77) (r28+#73) (r28+#81)    ;((r28+#77) plus (r28+#73))->(r28+#81)
1437 ???? 0000          movqi #1 (r28+#82)      ;#1->(r28+#82)
1438                    cmpqi (r28+#81) (r28+#73)       ;((r28+#81)-(r28+#73))->CC
1439                    bltu .L68
1440 ???? 0000          movqi #0 (r28+#82)      ;#0->(r28+#82)
1441            .L68:
1442                    iorqi3 (r28+#80) (r28+#82) (r28+#80)    ;((r28+#80) ior (r28+#82))->(r28+#80)
1443 ???? 0000          movqi (r28+#81) (r28+#73)       ;(r28+#81)->(r28+#73)
1444                    addqi3 (r28+#64) (r28+#68) (r28+#72)    ;((r28+#64) plus (r28+#68))->(r28+#72)
1445                    addqi3 (r28+#80) (r28+#72) r24  ;((r28+#80) plus (r28+#72))->r24
1446 ???? 0000          movqi r24 (r28+#72)     ;r24->(r28+#72)
1447 ???? 0000          movqi (r28+#72) (r28+#25)       ;(r28+#72)->(r28+#25)
1448 ???? 0000          movqi (r28+#73) (r28+#26)       ;(r28+#73)->(r28+#26)
1449 ???? 0000          movqi (r28+#74) (r28+#27)       ;(r28+#74)->(r28+#27)
1450 ???? 0000          movqi (r28+#75) (r28+#28)       ;(r28+#75)->(r28+#28)
1451            .L63:
```

Figure 53 Assembly listing of two lines of C source.

The assembly listing above corresponds to lines 240 and 241 taken from the C code in Figure 52.

- The first line begins with 240 and is succeeded by the name of the original source file. This line indicates that the line numbers 1396 to 1399 of the assembly listing were created when line 240 of `ADPCM_Verified.c` was compiled. The third parameter in the `.stabn 68,0,240,.LM74-adpcm_decoder` line also confirms this.
- According to the gcov logfile, line number 240 was executed 256 times when run dynamically and so it would be expected that the opcodes `addhi3`, `tsthi` and `beq` will also be executed 256 times if this assembly program was run.

Following on from this logic, it is possible to build a dynamic instruction count for the entire assembly output and to that end a Perl script was developed to count and summarize the

static and dynamic *insn* usage for a given application. As a result, it allows dynamic *insn* usage information to be produced for an application developed to run on a processor that is still under development.

```
Opcode Usage Statistics:
Opcode      Static   Dynamic
----------------------------
addhi3      58       8455
addqi3      5        1024
andhi3      9        2048
andqi3      3        512
ashlhi3     2        256
ashrhi3     6        1275
ashrqi3     5        642
beq         10       2305
bge         5        1280
bgt         2        514
ble         5        1281
blt         3        768
bne         1        1
call        2        2
cmphi       48       9864
cmpqi       16       3585
iorhi3      11       2427
iorqi3      1        128
jump        15       2180
lshrhi3     1        128
movhi       83       5870
movqi       105      12840
pophi       2        2
pushhi      2        2
return      2        2
seq         2        512
sgt         17       2946
sgtu        7        1792
sltu        12       2050
subhi3      24       4394

Insns: 30, Static: 464, Dynamic: 69085
```

Figure 54 Comparison of Static and Dynamic *insn* usage.

Dynamic Analysis Limitations

Using the methods described in the previous section to produce dynamic *insn* usage information from a static compilation is not without its limitations. Inaccuracies occur when a single, high-level source line compiles to assembly code with multiple execution paths. For example, Figure 53 reveals that line 241 in the original source code has resulted in 51 assembly lines being produced which contain 5 branches (`bltu` on lines 1413, 1419, 1425, 1433, & 1439). The execution of the `movqi` instruction that immediately succeeds each of these branch instructions will be dependent on whether the branch is taken or not. If all branches are in fact taken then the tally for the `movqi` instruction would be out by a factor of 5 from the true result. This failing is even further exacerbated by some of the code constructs available in high-level languages such as C.

Problems with `while()` Loops

When a `while()` loop is compiled into assembly code, a test and jump sequence is often produced:

```
343:/ADPCM_Verified.c ****     while (i < DATASIZE)
1274                   .stabn 68,0,343,.LM111-main
1275                .LM111:
1276                .L47:
1277                   cmphi (r28+#1) #255     ;((r28+#1)-#255)->CC
1278                   ble .L49
1279                   jump .L39
1280                .L49:
```

Figure 55 Assembly code generated from a `while()` loop.

An examination of the flow of execution reveals that the `ble` (branch if less than zero) instruction will be executed for each pass through the loop whilst the `jump` instruction will only be executed when the loop is exited. Because these assembly lines are all derived from a single high level source line, `gcov` does not resolve the two execution paths.

```
      257:  343:    while (i < DATASIZE)
```

Figure 56 `gcov` output for a `while()` loop.

According to `gcov`, the `while()` statement has been executed 257 times and this value would be added to the tallies for the `cmphi`, `ble` and `jump` instructions. This, of course, is incorrect as the `jump` instruction will only occur once.

To avoid introducing this sort of inaccuracy whilst still maintaining the program's logic, the `while()` loop of Figure 55 should be modified to that of Figure 57.

```
      257:  317:    while (1)
        -:  318:      {
      257:  319:     if (i >= DATASIZE)
        1:  320:       break;
```

Figure 57 `gcov` output for the alternate `while()` loop encoding.

In this case, the `while()` loop is coded as an infinite loop with the exit condition moved to a separated line within the loop. This gives `gcov` the ability to correctly resolve how many times the jump instruction is actually executed (see Figure 58).

```
317:/ADPCM_Verified.c ****     while (1)
318:/ADPCM_Verified.c ****       {
319:/ADPCM_Verified.c ****   if (i >= DATASIZE)
1232                   .stabn 68,0,319,.LM102-main
1233                .LM102:
1234                   cmphi (r28+#1) #255     ;((r28+#1)-#255)->CC
1235                   ble .L43
320:/ADPCM_Verified.c ****     break;
1236                   .stabn 68,0,320,.LM103-main
1237                .LM103:
1238                   jump .L41
1239                .L43:
```

Figure 58 Assembly listing of an alternate `while()` loop encoding (C code is interspersed with assembly code).

Problems with `for(;;)` Loops

```
      513:  109:    for (; len > 0; len--)
        -:  110:    {
      256:  111:        val = * inp++;
```

Figure 59 `gcov` output for a segment of a `for(;;)` loop.

`for(;;)` loops are another source of potential inaccuracies. According to the listing produced by `gcov` in Figure 59, line 109 has executed 513 times. One might initially conclude that the `for(;;)` loop as a whole has been executed 513 times however line 111 indicates that this is not correct. Line 111 has only executed 256 times and since this line is the first line in the loop, it would appear that the loop as a whole could only have executed 256 times.

Closer inspection reveals that the `for(;;)` loop actually contains three execution portions.

1. Initialization – if present will be executed once for the loop.
2. Test – if present will be executed at the beginning of each loop and will determine whether the body of the loop is to be entered.
3. Iteration – if present will be executed at the conclusion of each pass through the loop.

As a consequence, `gcov` will tally a `for(;;)` statement with all portions present as:

Initialization:	1	
Test:	n+1	
Iteration:	n	+
Total:	**2(n+1)**	where n = number of times the loop is entered.

In the example listing of Figure 59, there is no initialisation portion in the loop and so the loop statement will be executed 2(n+1) – 1 = 2(256+1) – 1 = 513 times which is consistent with the results recorded by `gcov` in Figure 59.

```
109:/../DynamicAnalysis/ADPCM_Verified.c ****     for (; len > 0; len--)
741                .stabn 68,0,109,.LM8-adpcm_coder
742              .LM8:
743              .L2:
744                cmphi (r28+#5) #0    ;((r28+#5)-#0)->CC
745                bgt .L5
746                jump .L3
747              .L5:

...                    ...

992                .stabn 68,0,109,.LM47-adpcm_coder
993              .LM47:
994                addhi3 (r28+#5) #-1 (r28+#5)     ;((r28+#5) plus #-1)->(r28+#5)
995                jump .L2
996              .L3:
```

Figure 60 Assembly listing of Test and Iteration portions of a `for(;;)` loop.

Figure 60 shows the assembly code listing for the Test and Iteration portions of the `for(;;)` loop in Figure 59. These two portions of code are separated by almost 250 assembly lines with the Test portion being executed at the beginning of the loop and the Iteration portion being executed at the end of the loop. A summary of the errors introduced as a result of incorrect tallying the assembly listing of Figure 60 is given as:

Instruction	Tally	Actual	Error
`cmphi`	513	257	256 (100%)
`bgt`	513	257	256 (100%)
`jump`	1026	257	769 (400%)
`addhi3`	513	256	257 (100%)

Table 1 Comparison of actual instruction counts versus tallied counts from Figure 60.

This level of error is significant but it is entirely avoidable if all `for(;;)` loops are replaced by infinite `while()` loops as per the previous discussion. The code of Figure 59 should therefore be re-coded according to Figure 61.

```
  -:  110:      //    for (; len > 0; len--)
257:  111:    while (1)
  -:  112:    {
257:  113:      if (len <= 0)
  1:  114:     break;
  -:  115:
256:  116:        val = * inp++;
...:  ...     ...
  -:  198:
256:  199:     len--;
  -:  200:    }
```

Figure 61 Replacing a `for(;;)` loop with a `while()` loop.

Problems with Conditional Assignment Constructs

The C language contains a conditional assignment construct that can be written on a single line such as:

```
256:  115:        sign = (diff < 0) ? 8: 0;
```

Figure 62 Example of a Conditional Assignment Construct.

The logic of Figure 62 is that `sign` would take the value `8` if `diff` is less than `0` otherwise `sign` would take the value `0`. The assembly listing produced from this code is given in Figure 63.

```
115:/../ADPCM_Verified.c ****         sign = (diff < 0) ? 8: 0;
760                     .stabn 68,0,115,.LM11-adpcm_coder
761                 .LM11:
762                     cmphi (r28+#18) #0      ;((r28+#18)-#0)->CC
763                     bge .L6
764 ???? 0000           movqi #8 (r28+#43)      ;#8->(r28+#43)
765                     jump .L7
766                 .L6:
767 ???? 0000           movqi #0 (r28+#43)      ;#0->(r28+#43)
768                 .L7:
769 ???? 0000           movqi (r28+#43) (r28+#15)     ;(r28+#43)->(r28+#15)
```

Figure 63 Assembly listing of Conditional Assignment Construct of Figure 62.

Once again the *insns* will be incorrectly tallied since `gcov` stores an execution tally for the entire line without regard for which branch is taken. The inaccuracy for the above assembly code in Figure 63 would be:

Instruction	Tally	Actual	Error
`cmphi`	256	256	0 (0%)
`bge`	256	256	0 (0%)
`jump`	256	0 - 256	0 - 256 (0 - 100%)
`movqi`	768	512	256 (50%)

Table 2 Comparison of actual instruction counts versus tallied counts from Figure 63.

Although this error is not as significant as the error demonstrated in the previous section, it is still avoidable. By expanding the conditional assignment into an if/else block that extends across multiple lines, `gcov` will be able to correctly determine the true instruction execution tally.

Concluding Remarks

This chapter has shown the modifications that were made to the GCC environment to adapt it to Compiler Directed Codesign. A mechanism for rapidly changing the machine description file as well as a means to efficiently extract dynamic *insn* usage information has been presented. Some limitations of this method were presented however so too were a number of minor code adjustments that can be made to the source application to overcome these limitations.

At this point in the dissertation, all of the core components of the Compiler Directed Codesign methodology have been described. The next chapter will use an Adaptive PCM Encoder/Decoder application as a case study to demonstrate the use of Compiler Directed Codesign in practice.

Chapter 6.

A CASE STUDY IN COMPILER DIRECTED CODESIGN

An Adaptive PCM Encoder/Decoder

The previous chapters have laid the foundations for Compiler Directed Codesign and described the platform that has been constructed to support this approach. In this chapter, I will demonstrate how the Compiler Directed Codesign methodology can be used to asses the cost/performance profile of a typical embedded application.

The chapter begins with a discussion of refinements that were made to the case study application in order to assist with the testing requirements. Following that, the process of using the Compiler Directed Codesign methodology to determine the final processor instruction set is explained. Finally the chapter concludes with a comparison between a processor produced as a result of the Compiler Directed Codesign methodology and a number of other processors.

Selecting a Test Application

The objective of this case study was to demonstrate the use of the Compiler Directed Codesign methodology. This was done by assessing the cost/performance profile of a *typical* program and using those results to direct the implementation of an FPGA-based *soft* processor capable of running the test application. A comparison could then be made between the performance of the custom processor and various commercially available soft-processors.

In considering the merits of selecting several applications versus selecting just one for the analysis, it was concluded that a single application would be sufficient to demonstrate the process of Compiler Directed Codesign and its impact on hardware/software co-design. A series of analyses conducted over a wide range of alternative applications was considered to be unnecessary and outside the scope of this work.

After some consideration, the test application was chosen from the MiBench test suite [136]. Full source code for all MiBench applications are available from [137]. Of the 35

applications available, the Adaptive PCM Encoder / Decoder (ADPCM) application was chosen for the following reasons:

1. It was considered to be sufficiently typical of current workloads to provide a valid demonstration of the Compiler Directed Codesign methodology and to provide evidence of the sort of benefits that are possible.
2. It only requires fixed point arithmetic. As it was important to be able to implement (in an FPGA) the processor derived from the case study, it was desirable to exclude the need to construct a floating point unit.
3. The size of the datasets within the application could be scaled easily to fit within the available memory of an FPGA device.
4. A self-test routine could be readily implemented to verify that the application had executed correctly.
5. The application has a high proportion of ALU operations compared with memory operations. This allows a broad range of operations to be exercised within the processor and lets the analysis focus on the computational capabilities of the processor rather than its ability to move data in and out of memory.

Preparing the Test Application

A number of minor modifications were made to the original ADPCM application to make it easier to test. A brief summary of the changes, and the reason for those changes, follows.

Code Additions

In its original form, the benchmark program consists of two function; one for encoding PCM data and one for decoding it. In order to create a single, self-contained application, a `main()` function was added which called the encode and decode routines.

In addition to this, the `main()` function also included a loop that compared the results of the encoding and decoding functions against a reference dataset. This verification step was necessary to ensure that the compiled code resulting from the encode and decode functions had been implemented correctly for each of the tested processors.

Resizing Variables

The original ADPCM application was coded using mainly 32-bit variables while the datasets only required 8-bit variables. This meant that several of the variables used throughout the application were much larger than the range of data that they needed to store. An

examination of all variables was made and their respective sizes were reduced to either 8-bit or 16-bit data types as applicable.

Code Transformations

In order to avoid the limitations of dynamic analysis that were identified in the previous chapter, all `for(;;)`, `while()`, and conditional assignment statements were transformed using the recommendations that were also presented in that chapter. A complete listing of the ADPCM routine is include in Appendix A.

Insn **Reduction Strategies.**

The goal throughout this case study was to explore the least complex processor capable of running the target application whilst still maintaining an adequate level of performance. To this end, some broad generalizations can be made:

- Processor complexity will *tend* to increase as the number of supported instructions is increased.
- Execution time will *tend* to increase as the code size of the application increases.

There are several exceptions to these broad statements however if we accept them as *trends* rather than as gospel, a broad brushed optimization strategy can be applied. This strategy will progressively decrease the number of unique *insns* whilst attempting to limit any increase that the reduction might have on the execution time of the application.

Currently, the process of selecting or rejecting *insns* is a manual process that is guided by the skill of an expert operator. The exact strategy employed will involve the balancing of several competing factors but will most likely involve one or more of the following considerations:

Decreasing the Range of Memory Modes.

This is the simplest *insn* reduction technique as it is only reliant on a static analysis of the *insn* mix used to execute the application. By decreasing the range of variable widths that the machine needs to process, the machine's datapath can be simplified to support only the most primitive data types. This also simplifies the machine's control path and instruction decoding/sequencing circuitry. In some circumstances it may be desirable to rationalize all *insns* to one or two memory modes as the memory interface and instruction encoding of the resultant processor will be simplified.

Removing Infrequently used Insns.

By removing support for *insns* that are used infrequently and replacing them with alternate *insn* sequences, processor resources can be conserved with minimal impact on the overall performance of the application. At first glance a static analysis may offer limited success in identifying infrequently used *insns* however this can not be considered reliable in the general case. Static analysis does not consider how many times a given block of code is executed when the application is running and can therefore produce misleading information.

The best way to identify *insns* of this nature is through a dynamic analysis of the code. *Insn* usage can then be tallied against one another to quickly reveal those that are used the least.

Remove High Resource Insns

By removing *insns* that require a high amount of hardware resources to implement and replacing them with a sequence of less complex *insns*, it may be possible to limit the complexity of the end processor without adversely impacting the overall performance. However, in order for this strategy to be deployed, silicon resource information is required for every *insn*.

Although this approach would allow a detailed cost/performance analysis to be made, it is heavily dependant on the implementation of the processor. Also, not all *insns* can be assessed in isolation from one another as some *insns* can be executed on a mixture of resources demanded by other *insns*. For example an 8-bit add would use a subset of hardware consumed by a 16-bit add. In this case the subset *insn* (8-bit add) is largely 'free' other than the additional resources required for decoding the *insn*.

Because of its high dependence on implementation, this approach to *insn* reduction was deemed outside of the scope of the thesis and not considered important to this investigation of the merit of Compiler Directed Codesign. It would, however, be a worthy extension topic.

Exhaustive Simulation

Exhaustive simulation brings nothing more to the table other than brute force. Rather than carefully considering the impact that each *insn* might have on the application, it simply simulates all *insn* combinations to try to identify the optimal set.

For exhaustive simulation to be effective, it would be necessary to include some level of automation but even if this was implemented, the simulation time would be significant. The ADPCM application that was analysed requested 38 *insns* when given total freedom of choice. Conducting an exhaustive simulation over these 38 *insns* would require

274,877,906,943 simulations (assuming no additional *insns* were added to the mixture). Currently it takes around 3 minutes to rebuild the compiler and recompile the application. At that rate it would take almost 1,568,938 years to simulate all combinations. This is clearly unacceptable and can be dismissed as a viable approach.

Combining Optimization Strategies

By utilizing a combination of the above optimization strategies the developer can have the best of all worlds. Although it requires far more user intervention and is more difficult to automate, an expert user can use their knowledge and experience to apply varying strategies throughout the optimization process. This balanced approach to optimization was the method deployed during this case study.

Insn Reduction Results

The purpose of this chapter is to present results obtained from using the Compiler Directed Codesign methodology on a *real world* design. The goal was to identify the least complex processor that is capable of running the target application without severely impacting on performance.

To begin the optimization process, a baseline was taken by first compiling the ADPCM application with all possible *insn* combinations enabled. This *blue sky* compilation is listed in the results as Iteration 0 and serves as the baseline against which all other iterations will be compared. At each iteration, I will discuss its results and indicate the reasoning for the actions taken for the next iteration. A complete summary of the results of all iterations is provided in Table 4 towards the latter half of this Chapter.

Iteration 0:

Setup: All *insns* enabled

Result:

Unique *insns*:	38
Static *insns*:	274
Dynamic *insns*:	34245

Discussion: A brief analysis of Figure 64 reveals that the compiler sought *insns* with data modes ranging across 8-bit (`qi`), 16-bit (`hi`) and 32-bit (`si`). The importance of taking a dynamic view of the application rather than a static one is readily apparent since the most frequently used dynamic *insn* (the `addhi3` *insn*) was only

the third most frequently used static *insn*. From this point on static *insn* usage statistics will be recorded merely as a point of interest however it is the dynamic *insn* statistics that will guide the iterative process.

```
Opcode Usage Statistics:
Opcode      Static   Dynamic
----------------------------
addhi3      26       3711
addqi3      2        512
addsi3      6        770
andhi3      3        512
andqi3      3        512
andsi3      3        768
ashlsi3     1        128
ashrhi3     6        1275
ashrqi3     5        642
beq         10       2305
bge         5        1280
bgt         2        514
ble         5        1281
blt         3        768
bne         1        1
call        2        2
cmphi       4        1025
cmpqi       3        768
cmpsi       4        1024
extendhisi2 14       2434
extendqihi2 3        512
iorhi3      3        507
iorqi3      1        128
jump        15       1412
movhi       54       2664
movqi       48       2848
movsi       8        514
neghi2      1        128
pophi       2        2
pushhi      2        2
return      2        2
seq         2        512
subhi3      5        426
subsi3      2        256
tsthi       4        771
tstqi       10       2305
tstsi       3        768
zero_extendqihi2     1  256

Insns: 38, Static: 274, Dynamic: 34245
```

Figure 64 *Insn* usage for baseline compilation of ADPCM application.

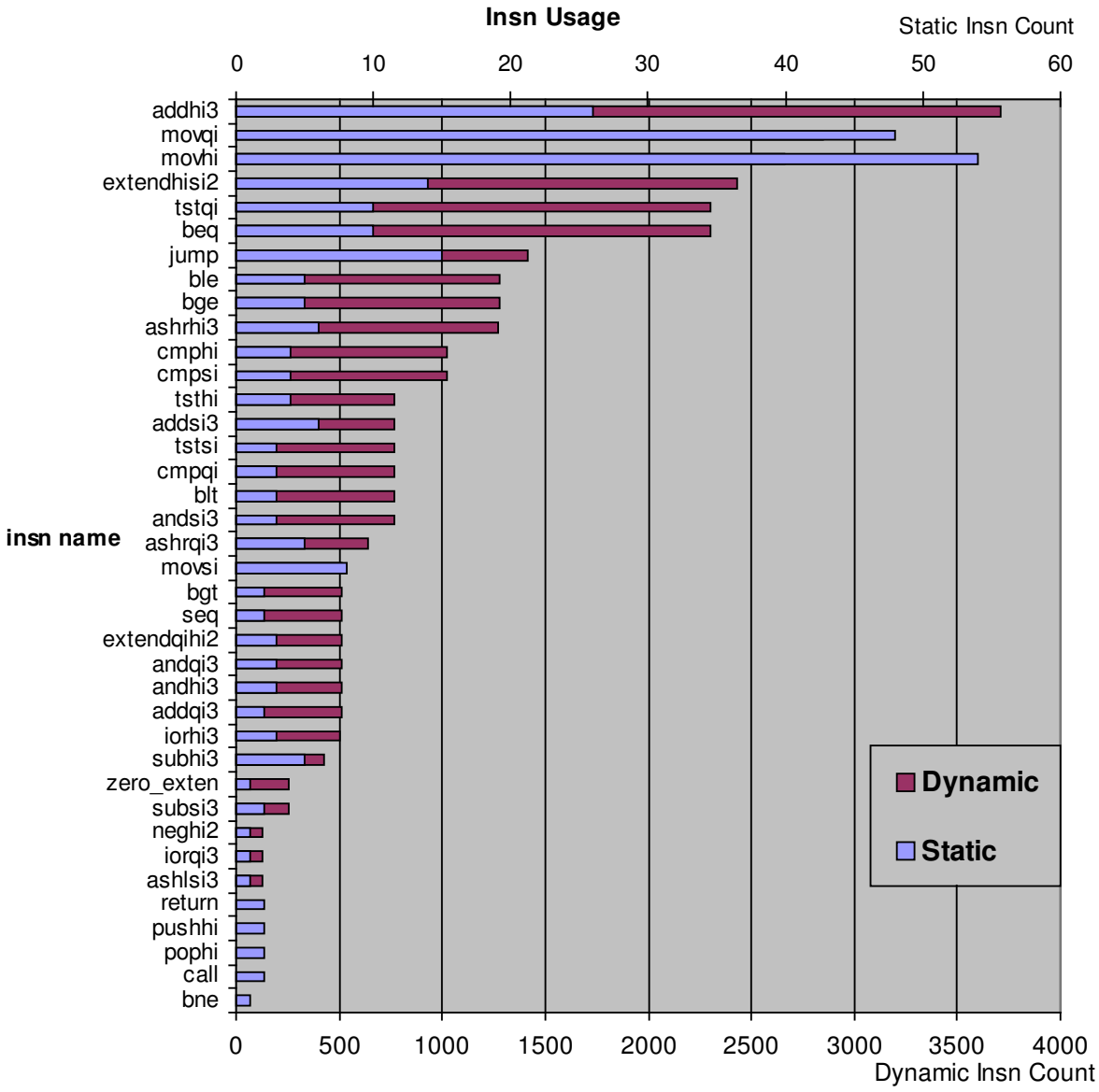


Figure 65 Tally of static and dynamic *insns* found in Iteration 0.

Iteration 1:

Setup: The initial reduction approach is to reduce the range of *insn* modes (bit widths) used. The largest *insn* mode present is the `SI` mode (32-bit) and so this mode will be targeted for removal first.

Disable all *insns* using a memory mode of `SI` (32-bit) or greater.

Disable `extendhisi` *insn* as this *insn* expands data to 32-bits.

Result: Unique *insns*: 38
Static *insns*: 632
Dynamic *insns*: 86541

Discussion: The compiler was not able to offer a substitution for the `ashlsi3` *insn* and subsequently replaced it with a `call_value` *insn* which invokes the library function `ashlsi3()`. The compiler assumes that this function will be supplied as part of the standard `libgcc` run-time library on the target processor. This assumption is invalid for this case study and an alternate substitution for `ashlsi3` will need to be made.

Iteration 2:

Setup: In order to remove the `call_value` *insn*, provide the compiler with access to the `ashlsi3` *insn* through an explicit expansion:

```
(define_expand "ashlsi3"
 [(set (match_operand:SI 0 "general_operand" "=g")
       (ashift:SI (match_operand:SI 1 "general_operand" "g")
                  (match_operand:SI 2 "general_operand" "g")))]
  ""
  "{
   int ShiftVal = INTVAL(operands[2]);
   int RS_Val;

   if (ShiftVal >= 16)
      {
      emit_insn(gen_movhi(custom_subword(operands[0], 0, SImode), const0_rtx));
      ShiftVal -= 16;
      if (ShiftVal >= 16)
         {
         emit_insn(gen_movhi(custom_subword(operands[0], 1, SImode), const0_rtx));
         }
      else
         {
         emit_insn(gen_ashlhi3(custom_subword(operands[0], 1, SImode),
                               custom_subword(operands[1], 0, SImode),
                               GEN_INT(ShiftVal) ));
         }
      }
   else
      {
      RS_Val = 16 - ShiftVal;
      /* Shift LIFT the HIGH word by the SHIFT amount (op[2]) */
      /*  Put the result into the HIGH word of the result operand */
      emit_insn(gen_ashlhi3(custom_subword(operands[0], 1, SImode),
                            custom_subword(operands[1], 1, SImode),
                            operands[2]) );

      /* Shift RIGHT the LOW word by 16 - SHIFT amount */
      /*  Temporarily put the result into the LOW word of the result operands */
      emit_insn(gen_lshrhi3(custom_subword(operands[0], 0, SImode),
                            custom_subword(operands[1], 0, SImode),
                            GEN_INT(RS_Val) ));

      /* OR the temp result with the HIGH WORD operand */
      emit_insn(gen_iorhi3(custom_subword(operands[0], 1, SImode),
                           custom_subword(operands[0], 1, SImode),
                           custom_subword(operands[0], 0, SImode) ));

      /* Finally LEFT SHIFT the LOW word and place into LOW result operand */
      emit_insn(gen_ashlhi3(custom_subword(operands[0], 0, SImode),
                            custom_subword(operands[1], 0, SImode),
                            operands[2]) );
      }
   DONE;
   }" )
```

Figure 66 Manual expansion of `ashlsi3` *insn* applied to Iteration 2

Result:

Unique *insns*:	39
Static *insns*:	626
Dynamic *insns*	85773

Discussion: The *insn* expansion above successfully removed any `call_value` *insns*.

The consequence of removing the `SI` mode in Iteration 1 has lead to a huge increase in the dynamic *insn* tally. The goal will now be to minimise the impact of removing this mode by making available to the compiler manual expansions

for the most commonly used `SI` mode *insns* found in Iteration 0. The *insns* to be targeted are listed below (Figure 67):

```
Opcode Usage Statistics:
Opcode      Static   Dynamic
----------------------------
extendhisi2 14       2434
cmpsi       4        1024
addsi3      6        770
andsi3      3        768
tstsi       3        768
movsi       8        514
subsi3      2        256
ashlsi3     1        128
```

Figure 67 *Insns* to be targeted for further expansion over coming iterations

Iteration 3:

Setup: Expand `extendhisi2` (dynamic usage: 2434) as this was the most heavily used `SI` mode *insn* found in the baseline.

```
(define_expand "extendhisi2"
 [(set (match_operand:SI 0 "general_operand" "=g")
       (sign_extend:SI (match_operand:HI 1 "general_operand" "g")))]
  ""
  "{
   operands[2] = gen_reg_rtx (HImode);
   operands[3] = gen_reg_rtx (QImode);

   emit_insn(gen_movhi(custom_subword(operands[0], 0, SImode), operands[1]));
   emit_insn(gen_tsthi(operands[1]));
   emit_insn(gen_sge (operands[3]));
   emit_insn(gen_zero_extendqihi2(operands[2],operands[3]));
   emit_insn(gen_addhi3(custom_subword(operands[0], 1, SImode),
                        constm1_rtx,
                        operands[2] ));
   DONE;
  }" )
```

Figure 68 Manual expansion of `extendhisi2` *insn* applied to Iteration 3

Result:
Unique *insns*: 40
Static *insns*: 637
Dynamic *insns*: 87949

Discussion: The purpose of inserting manual expansions is to provide the compiler with access to *insns* that it wants to use to compile the program. Ideally, providing these expansions should decrease the total *insn* count however in this case the expansion has lead to an increase in total *insns*. Because it has failed to produce the desired result (i.e. a reduction in the total *insn* count), it should be removed in the next iteration and an alternate *insn* chosen for expansion.

Iteration 4:

Setup: Remove the `extendhisi2` expansion inserted in the previous iteration as it failed to accomplish the desired objective – a reduction in static and/or dynamic *insns*.

Although the `cmpsi` (dynamic usage: 1024) *insn* was the next most heavily used *insn*, its expansion is reliant on the `tstsi` and `subsi3` *insns*. Because these *insns* were disabled as part of removing the `SI` mode and only one *insn* should be modified on each iteration, an alternate path is required.

One possibility would be to expand the `addsi3` (dynamic usage: 770) *insn* as this was the next most heavily *insn* used after the `cmpsi` *insn* however knowledge of GCC reveals a limitation in the way it automatically expands `mov` *insns*. If a `mov` *insn* is not available for a required mode, GCC automatically reverts to use `mov` *insns* with the mode of the processor's minimum word size even if larger mode `mov` *insns* (but still smaller than the originally requested *insn*) are available. In this case, the minimum word size is 8-bits (mode `QI`) and so GCC would accommodated the lack of a `movsi` *insn* by substituting it with four, `movqi` *insns* (8-bit move). Clearly a better solution can be achieved by offering an expansion of `movsi` into two `movhi` *insns* (16-bit move) and so this was the expansion that was used for the next iteration.

```
(define_expand "movsi"
[(set (match_operand:SI 0 "general_operand" "=g")
      (match_operand:SI 1 "general_operand" "g") ) ]
 ""
 "{
  emit_insn (gen_movhi(custom_subword(operands[0], 0, SImode),
                       custom_subword(operands[1], 0, SImode)));
  emit_insn (gen_movhi(custom_subword(operands[0], 1, SImode),
                       custom_subword(operands[1], 1, SImode)));
  DONE;
  }" )
```

Figure 69 Manual expansion of `movsi` *insn* applied to Iteration 4

Result:
Unique *insns*: 37
Static *insns*: 578
Dynamic *insns*: 82681

Discussion: Applying this expansion has lead to a reduction in the dynamic *insn* count of over 3000 from iteration 2.

Iteration 5:

Setup: Expand the `addsi3` (dynamic usage: 770) *insn* as it was the next most frequently used *insn* after the `cmpsi` *insn*.

```
define_expand "addsi3"
 [(set (match_operand:SI 0 "general_operand" "=g")
       (plus:SI (match_operand:SI 1 "general_operand" "g")
                (match_operand:SI 2 "general_operand" "g")))]
  ""
  "{
   operands[3] = gen_reg_rtx (HImode);
   operands[4] = gen_reg_rtx (QImode);

   emit_insn(gen_addhi3 (custom_subword(operands[0], 0, SImode),
                         custom_subword(operands[1], 0, SImode),
                         custom_subword(operands[2], 0, SImode) ));
   emit_insn(gen_sltu (operands[4]));
   emit_insn(gen_zero_extendqihi2(operands[3],operands[4]));
   emit_insn(gen_addhi3 (custom_subword(operands[0], 1, SImode),
                         custom_subword(operands[1], 1, SImode),
                         custom_subword(operands[2], 1, SImode) ));
   emit_insn(gen_addhi3 (custom_subword(operands[0], 1, SImode),
                         operands[3],
                         custom_subword(operands[0], 1, SImode) ));
   DONE;
   }" )
```

Figure 70 Manual expansion of `addsi3` *insn* applied to Iteration 5

Result: Unique *insns*: 37
Static *insns*: 468
Dynamic *insns*: 68567

Discussion: Applying this expansion has lead to a reduction in the dynamic *insn* count of over 17000 from iteration 2.

Iteration 6:

Setup: Expand the `tstsi` (dynamic usage: 768) *insn* as it was the next most frequently used *insn* after `addsi3` and required for the `cmpsi` expansion.

```
(define_expand "tstsi"
 [(set (cc0)
       (match_operand:SI 0 "general_operand" "g"))]
  ""
  "{
   operands[1] = gen_reg_rtx(HImode);

   emit_insn(gen_tsthi(custom_subword(operands[0], 0, SImode)));
   emit_insn(gen_sgtu(custom_subword(operands[1], 0, HImode)));
   emit_insn(gen_movqi(custom_subword(operands[1], 1, HImode), const0_rtx));
   emit_insn(gen_iorhi3(operands[1],
                        operands[1],
                        custom_subword(operands[0], 1, SImode)));
   emit_insn(gen_tsthi(operands[1]));
   DONE;
   }" )
```

Figure 71 Manual expansion of `tstsi` *insn* applied to Iteration 6

Result: Unique *insns*: 37
Static *insns*: 468
Dynamic *insns*: 68567

Discussion: Applying this expansion has not changed the number of static or dynamic *insns* from the previous iteration however it has provided of the *insns* necessary for expanding `cmpsi` (see iteration 8).

Iteration 7:

Setup: Expand the `subsi3` (dynamic usage: 256) *insn*. Although `subsi3` is not the next most frequently used *insn* after tstsi3, it is required for the `cmpsi` expansion.

```
(define_expand "subsi3"
 [(set (match_operand:SI 0 "general_operand" "=g")
       (minus:SI (match_operand:SI 1 "general_operand" "g")
                 (match_operand:SI 2 "general_operand" "g")))]
  ""
  "{
   operands[3] = gen_reg_rtx(HImode);
   emit_insn(gen_subhi3 (custom_subword(operands[0], 0, SImode),
                         custom_subword(operands[1], 0, SImode),
                         custom_subword(operands[2], 0, SImode) ));
   emit_insn(gen_sltu (custom_subword(operands[3], 0, HImode)));
   emit_insn(gen_movqi (custom_subword(operands[3], 1, HImode), const0_rtx));
   emit_insn(gen_subhi3 (custom_subword(operands[0], 1, SImode),
                         custom_subword(operands[1], 1, SImode),
                         custom_subword(operands[2], 1, SImode) ));
   emit_insn(gen_subhi3 (custom_subword(operands[0], 1, SImode),
                         custom_subword(operands[0], 1, SImode),
                         operands[3] ));
   DONE;
   }" )
```

Figure 72 Manual expansion of `subsi3` *insn* applied to Iteration 7

Result: Unique *insns*: 35
Static *insns*: 426
Dynamic *insns*: 63191

Discussion: Applying this expansion has continued to reduce the static and dynamic *insn* counts from iteration 2 as well as providing an *insn* necessary for the `cmpsi` expansion (see Iteration 8).

Iteration 8:

Setup: Now that the `tstsi` and `subsi` *insns* have been added, expand the `cmpsi` (dynamic usage: 1024) *insn*.

```
(define_expand "cmpsi"
 [(set (cc0)
       (compare (match_operand:SI 0 "general_operand" "g")
                (match_operand:SI 1 "general_operand" "g")))]
  ""
  "{
   operands[2] = gen_reg_rtx(SImode);
   emit_insn(gen_subsi3(operands[2],operands[0],operands[1]));
   emit_insn(gen_tstsi(operands[2]));
   DONE;
   }" )
```

Figure 73 Manual expansion of `cmpsi` *insn* applied to Iteration 8

Result: Unique *insns*: 34

Static *insns*: 418

Dynamic *insns*: 61143

Discussion: Applying this expansion has now reduced the dynamic *insn* count by over 24000 from iteration 2.

Iteration 9:

Setup: Expand the `andsi` (dynamic usage: 768) *insn* as it is the only remaining `SI` mode *insn* that was removed when the `SI` mode was disabled in iteration 1.

```
(define_expand "andsi3"
 [(set (match_operand:SI 0 "general_operand" "=g")
       (and:SI (match_operand:SI 1 "general_operand" "g")
               (match_operand:SI 2 "general_operand" "g")))]
  ""
  "{
   emit_insn(gen_andhi3 (custom_subword(operands[0], 0, SImode),
                         custom_subword(operands[1], 0, SImode),
                         custom_subword(operands[2], 0, SImode) ));
   emit_insn(gen_andhi3 (custom_subword(operands[0], 1, SImode),
                         custom_subword(operands[1], 1, SImode),
                         custom_subword(operands[2], 1, SImode) ));
   DONE;
   }" )
```

Figure 74 Manual expansion of `andsi` *insn* applied to Iteration 9

Result: Unique *insns*: 34

Static *insns*: 412

Dynamic *insns*: 59607

Discussion: Although the dynamic *insn* count is still well in excess of the figure produced in the base result, the impact of the initial removal of all 32-bit *insns* (`SI` mode) in iteration 1 has been minimized. As a side benefit, the number of unique *insns* have also been reduced and this will have the effect of decreasing the instruction decode logic in the target processor.

A snapshot of the *insn* statistics is provided in Figure 75:

```
Opcode Usage Statistics:
Opcode      Static   Dynamic
---------------------------
addhi3      44       6021
addqi3      2        512
andhi3      9        2048
andqi3      3        512
ashlhi3     2        256
ashrhi3     20       3709
ashrqi3     5        642
beq         10       2305
bge         5        1280
bgt         2        514
ble         5        1281
blt         3        768
bne         1        1
call        2        2
cmphi       4        1025
cmpqi       3        768
extendqihi2 3        512
iorhi3      11       2427
iorqi3      1        128
jump        15       1412
lshrhi3     1        128
movhi       86       6128
movqi       89       10788
neghi2      1        128
pophi       2        2
pushhi      2        2
return      2        2
seq         2        512
sgtu        7        1792
sltu        12       2050
subhi3      23       4266
tsthi       18       4355
tstqi       10       2305
zero_extendqihi2     7  1026

Insns: 34, Static: 412, Dynamic: 59607
```

Figure 75 *Insn* usage after 9 iterations of *insn* reduction

Iteration 10:

Setup: The fact that 34 unique *insns* were used in the previous iteration suggests that at least 6 bits will be required to encode instructions on this processor. Reducing the number of unique *insns* to below 32 will allow this to be further reduced to only 5 bits. Using fewer bits to encode the final instructions will potentially reduce the memory required to store the application code.

In moving into the next round of iterations, the strategy will shift from minimizing the dynamic *insn* count to minimizing the number of unique *insns*. The goal will therefore be to identify *insns* that can be easily substituted by other *insns* with minimal impact on the static or dynamic *insn* count. The first *insn* substitution will be for the `tsthi` *insn*. In iteration 9, this *insn* was used statically 18 times and dynamically 4355 times. Substituting it with the `cmphi` *insn* is a 1:1 substitution that should decrease the number of unique *insns* without affecting the static and dynamic *insn* counts.

```
(define_expand "tsthi"
 [(set (cc0)
       (match_operand:HI 0 "general_operand" "g"))]
  ""
  "{
   emit_insn(gen_cmphi(operands[0], const0_rtx));
   DONE;
  }" )
```

Figure 76 Manual expansion of `tsthi` *insn* applied to Iteration 10

Result: Unique *insns*: 33
Static *insns*: 412
Dynamic *insns*: 59607

Discussion: As predicted, the unique *insn* count has been reduced by one compared with the previous iteration however the static and dynamic *insn* counts have remained unchanged.

Iteration 11:

Setup: The previous iteration substituted the `tsthi` *insn* with zero effect on the static or dynamic *insn* tallies. Continuing this substitution process to include the `tstqi` will allow the `tst` *insn* to be totally removed with minimal effect on the *insn* tallies.

```
(define_expand "tstqi"
 [(set (cc0)
       (match_operand:QI 0 "general_operand" "g"))]
  ""
  "{
   emit_insn(gen_cmpqi(operands[0], const0_rtx));
   DONE;
  }" )
```

Figure 77 Manual expansion of `tstqi` *insn* applied to Iteration 11

Result: Unique *insns*: 32
Static *insns*: 412
Dynamic *insns*: 59607

Discussion: Once again the unique *insn* count has been reduced by one compared with the previous iteration however the static and dynamic *insn* counts have remained unchanged.

Iteration 12:

Setup: The `neghi2` *insn* can also be readily substituted with minimal impact on the *insn* tallies and will allow a further reduction in the number of unique *insns*.

```
(define_expand "neghi2"
[(set (match_operand:HI 0 "general_operand" "=g")
      (neg:HI (match_operand:HI 1 "general_operand" "g")))]
 ""
 "{
  emit_insn(gen_subhi3(operands[0],const0_rtx, operands[1]));
  DONE;
  }" )
```

Figure 78 Manual expansion of `neghi2` *insn* applied to Iteration 12

Result: Unique *insns*: 31
Static *insns*: 412
Dynamic *insns*: 59607

Discussion: Once again the unique *insn* count has been reduced by one compared with the previous iteration however the static and dynamic *insn* counts have remained unchanged.

Iteration 13:

Setup: Although the goal of reducing the unique number of *insns* to a value less than 32 has been accomplished at this point, there may yet be opportunities to reduce the processor's core logic by further reducing the number of unique *insns*. Unfortunately all 1:1 substitution opportunities have been exhausted and so any further substitutions will lead to an increase in the dynamic *insn* count and will have a greater impact on the application's performance. Only the user will be able to determine if this is acceptable given the constraints of their specific application however for the purposes of illustration and to ensure sufficient data has been obtained to fully asses the solution, further *insn* substitutions should be attempted.

When identifying *insns* for possible removal via substitution, preference should be given to those *insns* that are only represented in one memory mode as they offer the greatest potential to reduce processor complexity. A secondary consideration is also the impact on *insn* tallies that the substitution may have. Based on the *insn* usage statistics of the previous iteration, the two most eligible *insns* that can be readily expanded are `extendqihi2` (dynamic usage: 512, substitution ratio 1:4) and `zero_extendqihi2` (dynamic usage: 1026, substitution ratio 1:2). Substituting either of these *insns* will have the same net effect on the dynamic *insn* count. The choice is arbitrary but `zero_extendqihi2` was selected for substitution in the next iteration.

```
(define_expand "zero_extendqihi2"
 [(set (match_operand:HI 0 "general_operand" "=g")
       (zero_extend:HI (match_operand:QI 1 "general_operand" "g")))]
  ""
  "{
   emit_insn(gen_movqi(custom_subword(operands[0], 0, HImode), operands[1]));
   emit_insn(gen_movqi(custom_subword(operands[0], 1, HImode), const0_rtx));
   DONE;
   }" )
```

Figure 79 Manual expansion of `zero_extendqihi2` *insn* applied to Iteration 13

Result: Unique *insns*: 30
Static *insns*: 419
Dynamic *insns*: 60633

Discussion: As predicted, the unique *insn* count was reduced but at the cost of an increase in the dynamic *insns*.

Iteration 14:

Setup: As indicated in the previous discussion, `extendqihi2` is the next *insn* targeted for removal by substitution.

```
(define_expand "extendqihi2"
 [(set (match_operand:HI 0 "general_operand" "=g")
       (sign_extend:HI (match_operand:QI 1 "general_operand" "g")))]
  ""
  "{
   operands[2] = gen_reg_rtx (QImode);

   emit_insn(gen_movqi(custom_subword(operands[0], 0, HImode), operands[1]));
   emit_insn(gen_tstqi(operands[1]));
   emit_insn(gen_sge (operands[2]));
   emit_insn(gen_addqi3(custom_subword(operands[0], 1, HImode),
                        constm1_rtx,
                        operands[2] ));
   DONE;
  }" )
```

Figure 80 Manual expansion of `extendqihi2` *insn* applied to Iteration 14

Result: Unique *insns*: 30
Static *insns*: 428
Dynamic *insns*: 62169

Discussion: Although the `extendqihi2` *insn* was removed from the *insn* list, the `sge` *insn* was added; the net result being a zero reduction in the unique *insn* count. Once again the end user will need to determine for themselves whether this is acceptable given their specific constraints. For the purposes of this study, pursuing further optimization opportunities is still of interest and so the side effect of the `sge` *insn* being added is acceptable. The reason this is acceptable is because the `sge` *insn* is a simple extension to the existing *set-on-condition insns* that are already present

(seq, sgtu, sltu) and the condition that the sge needs to test for is already available as part of the *branch-on-condition* bge *insn*. The result is that supporting this insn at the hardware level should have minimal impact on the size of the processor. A snapshot of the *insn* statistics produced from this iteration is provided in Figure 81:

```
Opcode Usage Statistics:
Opcode      Static   Dynamic
----------------------------
addhi3      44       6021
addqi3      5        1024
andhi3      9        2048
andqi3      3        512
ashlhi3     2        256
ashrhi3     20       3709
ashrqi3     5        642
beq         10       2305
bge         5        1280
bgt         2        514
ble         5        1281
blt         3        768
bne         1        1
call        2        2
cmphi       22       5380
cmpqi       16       3585
iorhi3      11       2427
iorqi3      1        128
jump        15       1412
lshrhi3     1        128
movhi       86       6128
movqi       106      13352
pophi       2        2
pushhi      2        2
return      2        2
seq         2        512
sge         3        512
sgtu        7        1792
sltu        12       2050
subhi3      24       4394

Insns: 30, Static: 428, Dynamic: 62169
```

Figure 81 *Insn* usage after 14 iterations of *insn* reduction

Iteration 15:

Setup: Having successfully removed all traces of SI mode *insns* and having also limited the number of unique *insns* to a bare minimum, the focus of the optimisation strategy can now shift to removing HI mode *insns*. Unlike iteration 1 where it was possible to completely suppress the SI mode and still achieve a successful compile, GCC is unable to compile the program if a blanket ban on HI mode *insns* is made. The solution is therefore to iteratively remove the HI *insns* one by one to the point where the compiler is no longer able to proceed.

Of the 30 unique *insns* present in the previous iteration, 11 were of mode HI. Of these 11, only eight can be expanded further using relatively simple substitution

expansions. The substitution ratio of each of these eight *insns* is provided in Table 3. They have been ordered from least to most impact on the dynamic usage tally should the substitution be made.

Opcode	Dynamic Usage	Substitution Ratio	Dynamic Increase
pophi	2	1:2	2
pushhi	2	1:2	2
andhi3	2048	1:2	2048
iorhi3	2427	1:2	2427
movhi	6128	1:2	6128
subhi3	4394	1:4	13182
addhi3	6021	1:4	18063
cmphi	5380	1:8	37660

Table 3 Impact that further *insn* reductions will have on the total *insn* tally.

According to Table 3, the two most eligible *insns* for substitution are `pophi` and `pushhi` however these *insns* can not be removed from the compiler through modification of the machine description file alone. The reason for this is that both of these *insns* are hard coded into the function prologue/epilogues that have been defined for the compiler. Although it would be possible for this to be modified at the compiler level, it does not enhance the point of the case study. In order to remain consistent with the other iterations no changes will be made to the compiler at this point in the analysis and the `pophi` and `pushhi` will be kept. After `pophi` and `pushhi`, the next *insn* in line for expansion is `andhi3`.

```
(define_expand "andhi3"
 [(set (match_operand:HI 0 "general_operand" "=g")
       (and:HI (match_operand:HI 1 "general_operand" "g")
               (match_operand:HI 2 "general_operand" "g")))]
  ""
  "{
   emit_insn(gen_andqi3 (custom_subword(operands[0], 0, HImode),
                         custom_subword(operands[1], 0, HImode),
                         custom_subword(operands[2], 0, HImode) ));
   emit_insn(gen_andqi3 (custom_subword(operands[0], 1, HImode),
                         custom_subword(operands[1], 1, HImode),
                         custom_subword(operands[2], 1, HImode) ));
   DONE;
   }" )
```

Figure 82 Manual expansion of `andhi3` *insn* applied to Iteration 15

Result: Unique *insns*: 29

Static *insns*: 437

Dynamic *insns*: 64217

Discussion: Another unique *insn* has been successfully removed but at the cost of increased *insn* tallies. The reader will note that the dynamic *insn* tally has increased by exactly 2048 as predicted in Table 3.

Iteration 16:

Setup: Expand `iorhi3`.

```
(define_expand "iorhi3"
 [(set (match_operand:HI 0 "general_operand" "=g")
       (ior:HI (match_operand:HI 1 "general_operand" "g")
               (match_operand:HI 2 "general_operand" "g")))]
  ""
  "{
   emit_insn(gen_iorqi3 (custom_subword(operands[0], 0, HImode),
                         custom_subword(operands[1], 0, HImode),
                         custom_subword(operands[2], 0, HImode) ));
   emit_insn(gen_iorqi3 (custom_subword(operands[0], 1, HImode),
                         custom_subword(operands[1], 1, HImode),
                         custom_subword(operands[2], 1, HImode) ));
   DONE;
   }" )
```

Figure 83 Manual expansion of `iorhi3` *insn* applied to Iteration 16

Result: Unique *insns*: 28
Static *insns*: 448
Dynamic *insns*: 66644

Discussion: Once again another unique *insn* has been successfully removed but at the cost of increased *insn* tallies. The tally increase is 2427 as predicted in Table 3.

Iteration 17:

Setup: Expand `movhi3`.

```
(define_expand "movhi"
 [(set (match_operand:HI 0 "general_operand" "=g")
       (match_operand:HI 1 "general_operand" "g") ) ]
  ""
  "{
   emit_insn (gen_movqi(custom_subword(operands[0], 0, HImode),
                        custom_subword(operands[1], 0, HImode)));
   emit_insn (gen_movqi(custom_subword(operands[0], 1, HImode),
                        custom_subword(operands[1], 1, HImode)));
   DONE;
   }" )
```

Figure 84 Manual expansion of `movhi` *insn* applied to Iteration 17

Result: Unique *insns*: FAIL
Static *insns*: FAIL
Dynamic *insns*: FAIL

Discussion: GCC requires a `mov` *insn* to be available for the smallest address word size of the target processor so that it can do pointer manipulation. Since the custom processor dictates an address word of 16-bits, GCC requires the `movhi` *insn* to be present in order to be able to move 16-bit address words. Without this *insn*, the compiler has failed to compile the application.

Iteration 18:

Setup: Reinstate `movhi3`.

Expand `subhi3`.

```
(define_expand "subhi3"
 [(set (match_operand:HI 0 "general_operand" "=g")
       (minus:HI (match_operand:HI 1 "general_operand" "g")
                 (match_operand:HI 2 "general_operand" "g")))]
  ""
  "{
   operands[3] = gen_reg_rtx (QImode);

   emit_insn(gen_subqi3 (custom_subword(operands[0], 0, HImode),
                         custom_subword(operands[1], 0, HImode),
                         custom_subword(operands[2], 0, HImode) ));
   emit_insn(gen_sltu (operands[3]));
   emit_insn(gen_subqi3 (custom_subword(operands[0], 1, HImode),
                         custom_subword(operands[1], 1, HImode),
                         custom_subword(operands[2], 1, HImode) ));
   emit_insn(gen_subqi3 (custom_subword(operands[0], 1, HImode),
                         custom_subword(operands[0], 1, HImode),
                         operands[3] ));
   DONE;
   }" )
```

Figure 85 Manual expansion of `subhi3` *insn* applied to Iteration 18

Result: Unique *insns*: 29

Static *insns*: 514

Dynamic *insns*: 79820

Discussion: This expansion has actually lead to an increase in the number of unique *insns*. This is because the `subhi3` expansion makes use of the `subqi3` *insn* which was unused in previous iterations but the `subhi3` *insn* is also used twice in the function prologue/epilogue code. As discussed before in iteration 15, the prologue/epilogue code could be modified at the compiler level however in order to maintain a consistent operating environment across all iterations this was deemed unnecessary.

Each previous instance of the `subhi3` *insn* will expand to 3 additional *insns*. It can therefore be shown that of the 4394 `subhi3` *insns* previously found in iteration 14, 4392 will be expanded and the remaining 2 `subhi3` *insns* will be left

unexpanded in the prologue/epilogue code. The 79820 *insns* recorded from this iteration is equal to the 66644 *insns* in iteration 16 + 4392 converted `subhi3` *insns* x 3.

Iteration 19:

Setup: Expand `addhi3`.

```
(define_expand "addhi3"
 [(set (match_operand:HI 0 "general_operand" "=g")
       (plus:HI (match_operand:HI 1 "general_operand" "g")
                (match_operand:HI 2 "general_operand" "g")))]
  ""
  "{
   operands[3] = gen_reg_rtx (HImode);
   operands[4] = gen_reg_rtx (QImode);

   emit_insn(gen_addqi3 (custom_subword(operands[0], 0, SImode),
                         custom_subword(operands[1], 0, SImode),
                         custom_subword(operands[2], 0, SImode) ));
   emit_insn(gen_sltu (custom_subword(operands[0], 1, HImode)));
   emit_insn(gen_addhi3 (custom_subword(operands[0], 1, SImode),
                         custom_subword(operands[0], 1, SImode),
                         custom_subword(operands[1], 1, SImode) ));
   emit_insn(gen_addhi3 (custom_subword(operands[0], 1, SImode),
                         custom_subword(operands[0], 1, SImode),
                         custom_subword(operands[2], 1, SImode) ));
   DONE;
   }" )
```

Figure 86 Manual expansion of `addhi3` *insn* applied to Iteration 19

Result: Unique *insns*: FAIL
Static *insns*: FAIL
Dynamic *insns*: FAIL

Discussion: Similarly to the `movhi` *insn*, GCC requires the presence of the `addhi3` *insn* for pointer manipulation. Without it, the compiler fails to compile the application.

Iteration 20:

Setup: Expand cmphi.

```
(define_expand "cmphi"
 [(set (cc0)
       (compare (match_operand:HI 0 "general_operand" "g")
                (match_operand:HI 1 "general_operand" "g")))]
  ""
  "{
   operands[2] = gen_reg_rtx(HImode);
   emit_insn(gen_subhi3(operands[2],operands[0],operands[1]));
   emit_insn(gen_tsthi(operands[2]));
   DONE;
   }" )
```

Figure 87 Manual expansion of `cmphi` *insn* applied to Iteration 20

Previously the `tsthi` *insn* was expanded in iteration 10 to be substituted by a `cmphi` *insn*. In order to avoid a circular reference, the `tsthi` *insn* expansion needs to be updated.

```
(define_expand "tsthi"
 [(set (cc0)
       (match_operand:HI 0 "general_operand" "g"))]
  ""
  "{
   operands[1] = gen_reg_rtx(QImode);
   emit_insn(gen_tstqi(custom_subword(operands[0], 0, HImode)));
   emit_insn(gen_sgtu(operands[1]));
   emit_insn(gen_iorqi3(operands[1],
                        operands[1],
                        custom_subword(operands[0], 1, HImode)));
   emit_insn(gen_tstqi(operands[1]));
   DONE;
  }" )
```

Figure 88 Manual expansion of `tsthi` *insn* applied to Iteration 20

Result: Unique *insns*: 27
Static *insns*: 596
Dynamic *insns*: 100060

Discussion: The unique *insn* count has progressively decreased however the dynamic *insn* count has dramatically increased. Because of the change in the `tsthi` expansion, the results predicted in Table 3 can no longer be relied upon.
At this point all of the simple substitution expansions that are available have been exhausted. A tabular and graphical summary of the insn tallies taken from all 20 iterations is listed in Table 4 and Figure 89.

Summary of Results

	Iteration Number																				
Insn	**Base**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
addhi3	3711	3711	3711	6145	3711	6021	6021	6021	6021	6021	6021	6021	6021	6021	6021	6021	6021		6021		6021
addqi3	512	5902	5902	5902	5902	512	512	512	512	512	512	512	512	512	1024	1024	1024		1024		1024
addsi3	770	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
andhi3	512	512	512	512	512	512	512	512	512	2048	2048	2048	2048	2048	2048	-	-		-		-
andqi3	512	3584	3584	3584	3584	3584	3584	3584	3584	512	512	512	512	512	512	4608	4608		4608		4608
andsi3	768	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
ashlhi3	-	-	256	256	256	256	256	256	256	256	256	256	256	256	256	256	256		256		256
ashlqi3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
ashlsi3	128	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
ashrhi3	1275	3709	3709	1275	3709	3709	3709	3709	3709	3709	3709	3709	3709	3709	3709	3709	3709		3709		3709
ashrqi3	642	642	642	642	642	642	642	642	642	642	642	642	642	642	642	642	642		642		642
beq	2305	2305	2305	2305	2305	2305	2305	2305	2305	2305	2305	2305	2305	2305	2305	2305	2305		2305		2305
bge	1280	768	768	768	768	768	768	768	1280	1280	1280	1280	1280	1280	1280	1280	1280		1280		1794
bgt	514	1026	1026	1026	1026	1026	1026	1026	514	514	514	514	514	514	514	514	514		514		-
bgtu	-	1536	1536	1536	1536	1536	1536	1536	-	-	-	-	-	-	-	-	-		-		-
ble	1281	769	769	769	769	769	769	769	1281	1281	1281	1281	1281	1281	1281	1281	1281		1281		512
blt	768	1280	1280	1280	1280	1280	1280	1280	768	768	768	768	768	768	768	768	768		768		1537
bltu	-	1024	1024	1024	1024	1024	1024	1024	-	-	-	-	-	-	-	-	-		-		-
bne	1	2561	2561	2561	2561	2561	2561	2561	1	1	1	1	1	1	1	1	1		1		1
call	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2		2		2
call_value	-	128	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
cmphi	1025	1025	1025	1025	1025	1025	1025	1025	1025	1025	5380	5380	5380	5380	5380	5380	5380		5380		-
cmpqi	768	9994	9994	9994	9994	6144	6144	4864	768	768	768	3073	3073	3073	3585	3585	3585		3585		14345
cmpsi	1024	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
extendhisi2	2434	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
extendqihi2	512	512	512	512	512	512	512	512	512	512	512	512	512	512	-	-	-		-		-
iorhi3	507	507	635	635	635	635	635	635	2427	2427	2427	2427	2427	2427	2427	2427	-		-		-
iorqi3	128	4484	4484	4484	4484	2944	2944	2432	128	128	128	128	128	128	128	128	4982		4982		10362
jump	1412	2436	2436	2436	2436	2436	2436	2436	1412	1412	1412	1412	1412	1412	1412	1412	1412		1412		1412
lshrhi3	-	-	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128		128		128
lshrqi3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
movhi	2664	5100	5100	4842	8176	7152	7152	6128	6128	6128	6128	6128	6128	6128	6128	6128	6128		6128		6128
movqi	2848	18866	17714	12846	11562	9252	9252	8740	10788	10788	10788	10788	10788	12840	13352	13352	13352		13352		13352
movsi	514	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
neghi2	128	128	128	128	128	128	128	128	128	128	128	128	-	-	-	-	-		-		-
neg_tsthi	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
pophi	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2		2		2
popqi	-	8	8	8	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
pushhi	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2		2		2
pushqi	-	8	8	8	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
return	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2		2		2
seq	512	512	512	512	512	512	512	512	512	512	512	512	512	512	512	512	512		512		512
sge	-	-	-	2434	-	-	-	-	-	-	-	-	-	-	512	512	512		512		512
sgt	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
sgtu	-	1280	1280	1280	1280	1280	1280	-	1792	1792	1792	1792	1792	1792	1792	1792	1792		1792		7172
sltu	-	3850	3850	3850	3850	770	770	1026	2050	2050	2050	2050	2050	2050	2050	2050	2050		6442		7467
subhi3	426	426	426	426	426	426	426	1194	4266	4266	4266	4266	4394	4394	4394	4394	4394		2		2
subqi3	-	1792	1792	1792	1792	1792	1792	-	-	-	-	-	-	-	-	-	-		13176		16251
subsi3	256	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
tsthi	771	771	771	3205	771	771	771	771	4355	4355	-	-	-	-	-	-	-		-		-
tstqi	2305	5121	5121	5121	5121	5121	5121	5121	2305	2305	2305	-	-	-	-	-	-		-		-
tstsi	768	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		-		-
zero_extendqihi2	256	256	256	2690	256	1026	1026	1026	1026	1026	1026	1026	1026	-	-	-	-		-		-
Total	**34245**	**86541**	**85773**	**87949**	**82681**	**68567**	**68567**	**63191**	**61143**	**59607**	**59607**	**59607**	**59607**	**60633**	**62169**	**64217**	**66644**	**0**	**79820**	**0**	**100060**
Unique *Insns*	**38**	**38**	**39**	**40**	**37**	**37**	**37**	**35**	**34**	**34**	**33**	**32**	**31**	**30**	**30**	**29**	**28**	**0**	**29**	**0**	**27**

FAIL

FAIL

Table 4 Summary of *insn* tallies taken across all 20 iterations.

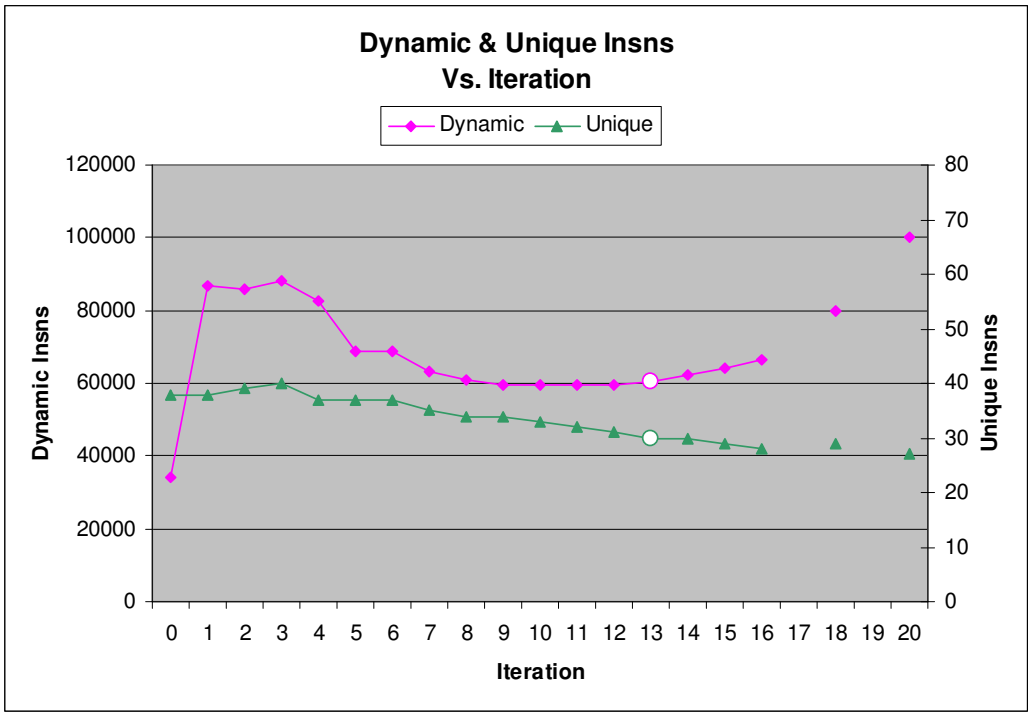


Figure 89 Graph of Dynamic *insn* tally and unique *insn* count from all 20 iterations

Figure 89 shows that the decision to remove the `SI` mode *insns* in Iteration 1 resulted in a marked increase in the dynamic *insn* count. Through the process of successive *insn* expansions and substitutions it was possible to reduce the impact of this decision through iterations 2 to 12. After that, the *insn* count began to steadily rise as further *insns* were removed or substituted. Beyond Iteration 16, the compiler began to fail as there were insufficient *insns* available for it to complete the compilation process. Where compilation was still possible at Iterations 18 and 20, the dynamic *insn* tally showed an upwards trend that suggested further benefits through *insn* reductions would be unlikely.

Determining the Final Solution

The objective of this case study was to determine the instruction set of the least complex processor capable of running the target ADPCM application without severely impacting on its performance. Having used the Compiler Directed Codesign methodology to perform 20 *insn* reduction iterations, the challenge of determining which iteration represents the *best* instruction set still remains.

Figure 89 plots both the dynamic *insn* count as well as the number of unique *insns* for each iteration. If we first consider the processor complexity requirement, this goal can be

translated to one of minimizing the number of unique *insns*. This occurs at iteration 20 but with the program requiring 100,060 *insns* to execute, this is likely to be in the order of almost 3 times slower than the baseline which only requires 34245 *insns*. Whether or not this impact on performance is acceptable will be entirely dependent on the constraints of the specific application.

The second requirement was that performance should not be severely impacted so on that basis the best solution would occur when the dynamic *insn* count is at a minimum. Iterations 9 through to 12 all required 59607 dynamic *insns* to execute and these are the lowest counts after the baseline. Given that iteration 12 only requires 31 unique *insns* however would suggest it to be the best solution.

But what of the iterations after iteration 12? Iteration 13 requires 1026 more *insns* to execute than iteration 12 but it uses one less unique *insn*. Furthermore, for a penalty of 2562 dynamic *insns*, iteration 14 provides a solution with two less unique *insns* than iteration 12.

It is difficult to tell from Figure 89 alone which of these, or any other, iterations is the best solution without some way of combining the relative merits of reducing the unique *insn* count against the cost of increasing the dynamic *insn* count.

Devising a Quality Factor to determine the Final Solution

The unique and dynamic *insn* counts work in competition to one another after Iteration 13 since as the unique *insn* count goes down, it has a negative impact on the dynamic *insn* count. At this point it must be emphasized that each application will have a different set of constraints and so it is difficult to propose a generalized solution for determining the best instruction set solution. However, for the purpose of exploring the results further, it is helpful to somehow combine the two competing forces into a single number that measures the relative improvement offered by each iteration. By plotting this *Quality Factor* (Q_f) for each iteration on a graph, it will be possible to quickly locate the solution.

In truth, any number of functions could be devised that linked together performance and complexity into a single figure but in this case, a fairly simple one will be sufficient. In Equation 2, an increased Q_f value will result if either the dynamic or unique *insn* tallies decrease with respect to the baseline (Iteration 0).

$$\text{Quality Factor (Qf)} = \frac{\text{Dynamic } \textit{Insn} \text{ Count of Base}}{\text{Dynamic } \textit{Insn} \text{ Count of Iteration}} \text{ X } \frac{\text{Unique } \textit{Insn} \text{ Count of Base}}{\text{Unique } \textit{Insn} \text{ Count of Iteration}}$$

Equation 2 Quality Factor used to determine the final solution.

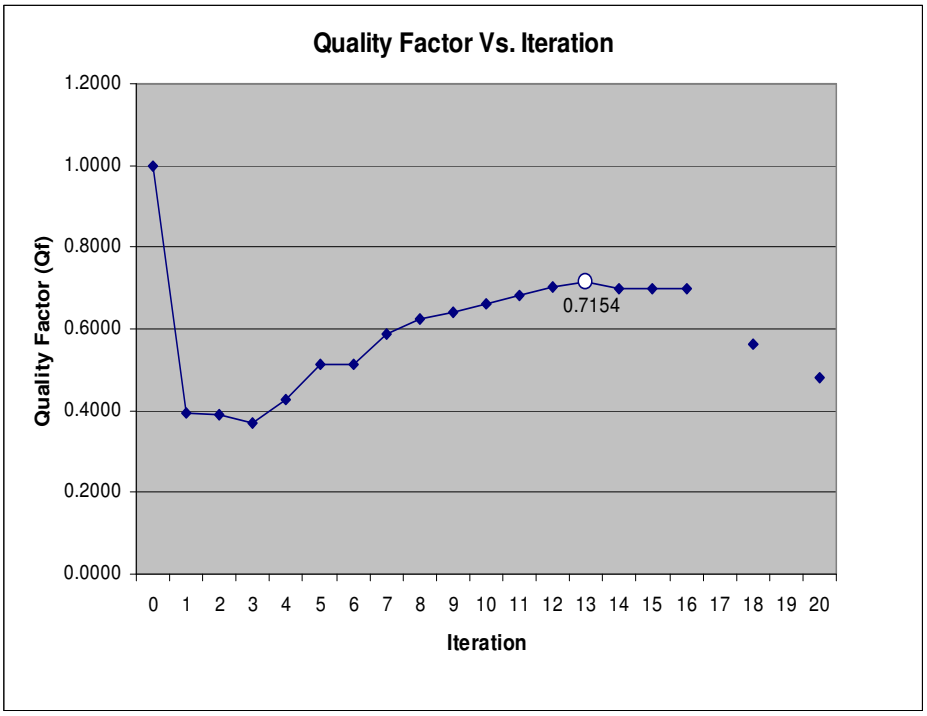


Figure 90 Quality Factor values across all iterations.

Using this simple measure, it is much easier to see that Iteration 13 stands out as a local maximum and therefore represents an ideal balance point between processor complexity and application performance for this case study.

The determination of the above quality factor is somewhat arbitrary as it doesn't take into account the relative cost that each instruction adds to the processor area. This cost can be measured in terms of raw number of instructions, instruction complexity, and whether the instruction requires additional resources or can make use of resources that have already been included in the processor. For example, decreasing the number of unique instructions from 33 down to 32 may represent a worthwhile cost saving since the number of bits required to represent the opcode can be reduced from 6-bits down to 5. However once the instruction count is below 32, it would require the removal of 16 further instructions before impacting on the opcode size. In this case, the benefit of reducing instructions would have to be weighed against the possible decrease in complexity of the processor.

Alternatively, the resources consumed by a multiplication instruction would presumably be much greater than those of an adder. On the other hand, if a subtract instruction has already been included then the resources required to also include a compare instruction are almost negligible and may be effectively free.

So to reiterate the point made before, the above determination of a quality factor was made purely on the basis of the relative contribution that each iteration made to the number of unique instructions and the dynamic instruction count. In future, it is hoped that this quality factor could be better informed with resource consumption information as well.

Verifying Compiler Directed Codesign

At this point the usage of Compiler Directed Codesign has been demonstrated using the ADPCM application as a case study and an instruction set solution has been proposed. However in order to further validate the methodology it is necessary to follow the development path through to the final production of a processor core that can be run on an FPGA and compared against alternate cores. This will then close the analysis loop and provide an opportunity to verify the dynamic insn predictions that were so heavily relied upon during the insn reduction process.

Developing a processor core is not a trivial task and could almost be considered a research project in itself. It requires careful consideration of a number of complex issues such as instruction encoding, data packing, word sizes, and micro sequencing. There are almost infinite possible combinations that could be chosen and a poorly implemented processor could quickly overwhelm any benefits offered by the Compiler Directed Codesign methodology.

Notwithstanding this, a complete processor core was developed in VHDL, simulated under ModelSim [138], synthesized and executed using Altium Designer [139]. The remainder of this chapter will discuss how this processor was developed and then used to verify the accuracy of the *insn* predictions.

Assembly output from CDC system	Adjusted assembly listing

`.file "ADPCM_Verified.c"`
`.arch custom2`
`.global __do_copy_data`
`.global __do_clear_bss`
`.global __time`
`.global __time`
`.section .bss`
`.type __time, @object`
`.size __time, 2`
`__time:`
`.skip 2,0`
`.global RefPCMdata`
…
	`.file "ADPCM_Verified.c"`
	`.arch custom2`
	`.org 0`
	`jump main`
	`; .global __do_copy_data`
	`; .global __do_clear_bss`
	`;.global __time`
	`;.global __time`
	`; .section .bss`
	`; .type __time, @object`
	`; .size __time, 2`
	`;__time:`
	`; .skip 2,0`
	`.global RefPCMdata`
	…

…
`main:`
`/* prologue: frame size=4 */`
`movhi (__stack - 4) r28`
`movhi r28 {__SP__}`
`/* prologue end (size=7) */`
…
	…
	`main:`
	`/* prologue: frame size=4 */`
	`; movhi (__stack - 4) r28`
	`movhi #0xFFFB r28`
	`movhi r28 {__SP__}`
	`/* prologue end (size=7) */`
	…

Legend:

Lines commented out

Lines manually added

Figure 91 Manual updates required before the assembly listing could be assembled.

Assembling the Test Program

The output of the Compiler Directed Codesign system is an assembly code listing which still needs to be assembled into the machine code of the target processor before it can be run. Clearly (by definition) there is no pre-existing assembler for the custom processor. Thus one was constructed using the PERL scripting language. The output of this custom assembler was a VHDL testbench that encapsulated and populated a program memory block and managed the simulation process. A complete listing of the PERL script is included in the electronic appendix.

Although the goal was to be able to take the compiled, assembler output from the Compiler Directed Codesign system and assemble it with minimal manual intervention, this was not fully achieved. Due to limitations in the PERL script, two minor but nonetheless manual

modifications were necessary before the Compiler Directed Codesign output could be assembled.

Referring to Figure 91, the `.org` and `jump` statements locate the program in memory and ensure that processing jumps to the main routine. And the `movhi #0xFFFB r28` line removes the need for the assembler to calculate the `(__stack - 4)` value and allows the stack pointer to be correctly initialised.

Constructing the Custom Processor

Compiler Directed Codesign utilizes the intermediate instructions within GCC to take a *compiler's view* of which resources it would like to see in the target processor. Under normal circumstances, these intermediate instructions would never be visible outside of GCC, but would be mapped to the assembly mnemonics of the target platform as part of the final phase of the compilation process. However, as the target platform is still under development at this point and no assembler mnemonics exist, the intermediate instructions used within GCC become the assembly commands of the processor platform being codesigned.

The processor developed to support these intermediate instructions will appear quite similar to any other processor in that it will have a set of registers, a sequencer and ALU. The implementation is entirely at the user's discretion but in this case study, the processor was given a block of 32, 8-bit general purpose registers, an ALU designed to support a select range of operations, and a sequencer unit responsible for co-ordinating the fetch/decode/execute cycles of the processor.

Because the assembler was developed specifically for the purposes of this case study, it produced a VHDL testbench file as its output rather than the traditional hex file. This testbench contained a memory entity that was preloaded with the application code as well as the main testbench driver code that reset the processor on startup and asserted the clock signal during simulation. A top level schematic of the complete system is provided in Figure 92.

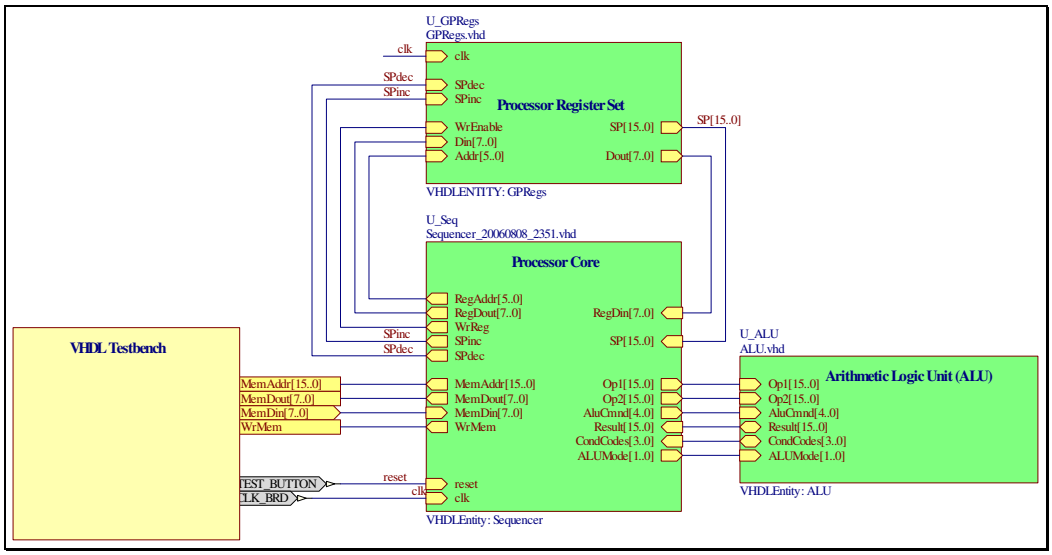


Figure 92 Top-level view of the processor simulation system

The main processor components (Figure 92) are coloured Green and consist of the Processor Core, the General Purpose Register block and the Arithmetic Logic Unit. The Yellow blocks represent the VHDL output produced by the custom assembler. There are two primary entities in this block; a memory block that has been preloaded with the program code, and some testbench management logic used to run the simulation. A complete listing of the VHDL code of the processor core is included in the electronic appendix.

Simulation Results

The entire system shown in Figure 92 was simulated using ModelSim. In addition to the processor's core logic, a number of counters were inserted into the processor core to count how often each instruction was used. An `ErrorCount` variable was used within the application and was incremented if the output of the ADPCM routines produced unexpected results. This variable was made visible through the simulator to allow quick validation of the program's execution.

The final results of the simulation are shown in Figure 93. The left column lists the `ErrorCount` variable at the top followed by the different instructions (*insns*). The next column to the right of this lists the value of the `ErrorCount` variable followed by the tally of how many times each instruction was executed during the running of the application. It can be seen that the `ErrorCount` variable is 0 indicating a valid result and the instruction counts match exactly with the dynamic *insn* predictions listed in iteration 13 of Table 4. This validates the predictive methods used by the Compiler Directed Codesign methodology to

inform the codesign process and demonstrates that dynamic insn information can be accurately calculated before the processor platform exists.

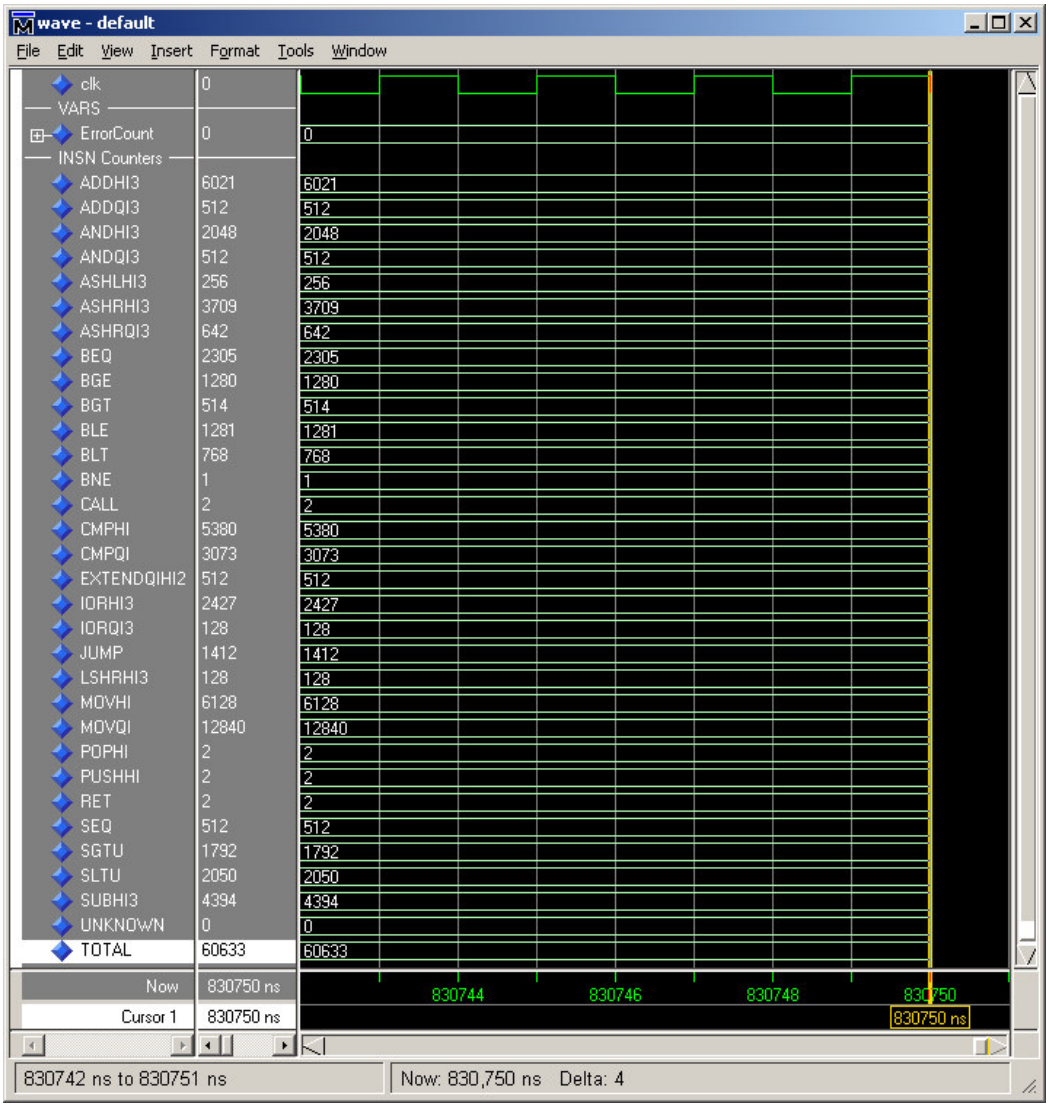


Figure 93 ModelSim results collected from the final stages of simulation.

Note that `ErrorCount` is 0 and that the individual instruction tallies are exactly the same as those predicted in Iteration 13 of Table 4.

Synthesis Results

Having verified the operation of the custom processor core through simulation, the final task was to synthesize the core and run it on an FPGA-based hardware platform. This would allow a comparison to be made between the custom processor core and other commercially available cores. Data relating to the FPGA resource usage and final execution speeds could then be compared.

A Xilinx Spartan3-1500 device was targeted as the host FPGA device and a summary of the resources consumed on this FPGA by the custom processor are included in Table 5. The key figures of merit useful to this discussion are the total number of 4-input LUT resources and the number of slice registers. A Look Up Table (LUT) is somewhat like a small memory cell and is what is used as the combinatorial unit in an FPGA. Whereas ASIC devices will measure complexity using gates as the metric, FPGAs use LUTs. The Spartan3-1500 has 13,312 LUTs distributed across 3,328 Configurable Logic Blocks (CLBs). In addition to this, each LUT is connected to a slice register which can be used for storing state information. The combination of LUT and slice registers are the common metrics used to indicate the size of a given circuit.

Altium Designer is a unified design system marketed by Altium Limited. It not only includes FPGA development tools, but also a number of processor cores that can be downloaded and run inside an FPGA. These cores include the TSK51 [140] which is modelled on Intel's 8051 processor, the TSK80A [141] which is modelled on Zilog's Z80, the TSK3000 [142] which is modelled on the MIPS-3000 core, and a variant of Xilinx's proprietary Microblaze [11].

Table 5 provides a comparison of the custom core that was developed as part of this thesis with the processor cores available in Altium Designer. A number of other metrics are also included indicating the FPGA resources consumed by these cores, the execution time, and the memory space used by the ADPCM program when run on each of these processors.

Processor	4-input LUT	Slice Registers	Execution Time (10MHz Clock)	Program Size (Bytes)	Comment
Custom	918	193	41.554ms	2565	Custom 8/16-bit processor
TSK51*	1385	296	455ms	3075	8051-like core with peripherals removed
TSK80A	2470	333	166.6ms	3423	Z80-like core (no peripherals)
TSK3000A	2859	1021	5.297ms	2844	MIPS3000-like core (no peripherals)
MicroBlaze	2514	1460	6.831ms	3649	Microblaze inside wishbone wrapper

Table 5 Comparison of Custom processor size and performance with commercially available processor cores

Resource Usage Comparison: 4-Input LUT

The 4-input LUT data is a measure of the logic resources used by each processor and was taken from summary information provided at the conclusion of the FPGA build flow or from documentation supplied by Altium [143]. Of the 4 processors listed in Table 5, the

TSK51 was the only one to include a number of peripherals as part of the core. In order to determine the resources used by just the core alone, the size of the timer and UART peripherals [144] was subtracted from the size of the overall processor.

It should also be noted that in order to maintain consistency across tests, the MicroBlaze variant used for testing was the one supported by Altium Designer. This particular variant includes a wishbone wrapper to allow for compatibility with other Altium peripherals. This wrapper increases FPGA resource consumption and adds some latency to bus accesses. According to Xilinx's own documentation [125], a lean version of the MicroBlaze will only consume 1324 LUTs on a Spartan3 and can run at speeds (peripheral permitting) up to 115MHz.

A comparison of the resources used by each of the cores is provided in Figure 94.

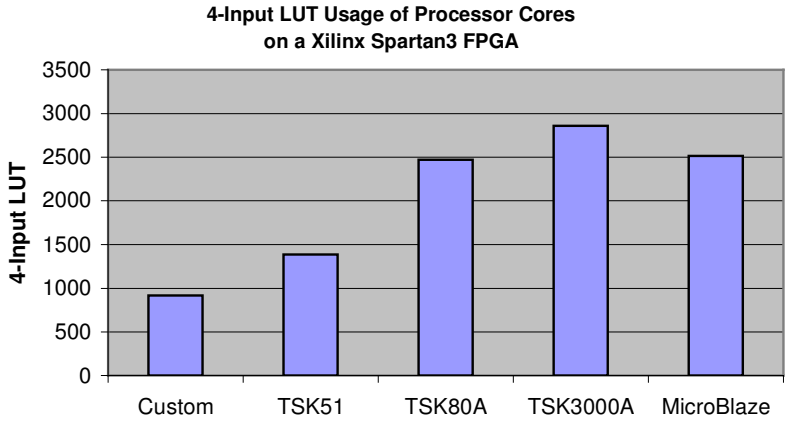


Figure 94 Comparison of 4-Input LUT usage on Custom processor and commercially available processor cores.

Resource Usage Comparison: Slice Registers

The slice register figure is a measure of how many FPGA registers (flip-flops) are used by the processor core. As with the 4-input LUTs, the registers used by the peripherals in the TSK51 were subtracted from the processor's overall figure to provide a better comparison. A comparison of the slice registers used by each of the cores is shown in Figure 95.

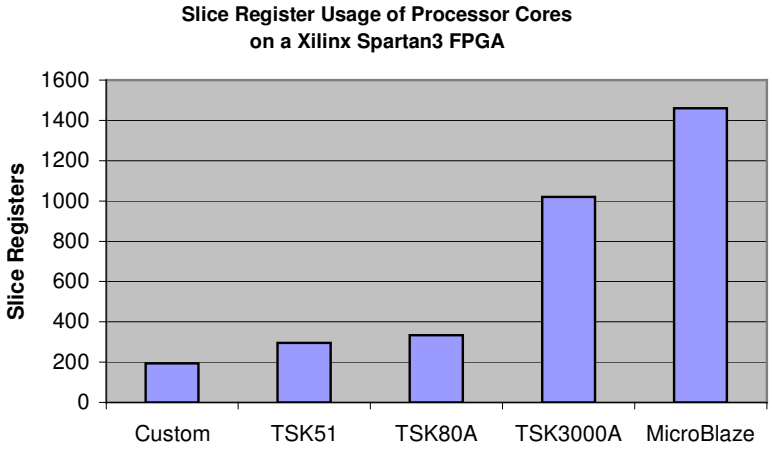


Figure 95 Comparison of Slice Register usage on Custom processor and commercially available processor cores.

Performance Comparison

The ADPCM application was run as an infinite loop on each of the tested processors. By toggling an output port each time the ADPCM application was called, a frequency counter instrument was then used to measure how long the application took to execute. A comparison between the different processors is provided in Figure 96.

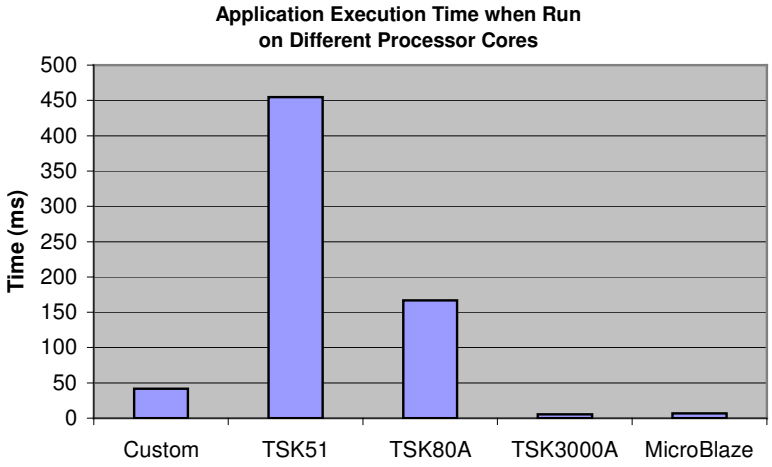


Figure 96 Comparison of Execution Time on Custom processor and commercially available processor cores.

Code Size Comparison

The application code size was taken from the hex file produced after the ADPCM application was compiled for each of the different processors. This metric provides an indication of each processor's code efficiency. A comparison is provided between the four processors in Figure 97.

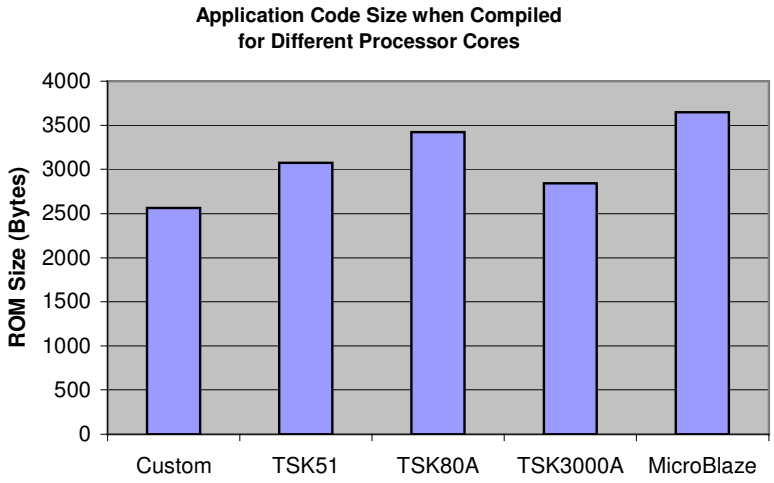


Figure 97 Comparison of Code Size on Custom processor and commercially available processor cores.

Summary

Although considerable time was expended developing the system architecture and processor core, very little effort was applied *optimising* the architecture. This was because the purpose of the exercise was to test the accuracy of the Compiler-Directed Codesign methodology rather than to test the author's ability to create a heavily optimised processor core.

However in light of this, it is still worth noting how favourably the custom processor core compares with the other commercially available cores. The custom core is 64% of the size of the next largest core and less than 32% of the size of the 32-bit TSK3000A core. Also the custom core requires less code memory to represent the ADPCM application than all of the other processors.

Internally, the custom processor had a 16-bit ALU but all memory IO operations were byte-wide. Its execution speed was almost 11 times faster than the TSK51 and over 4 times faster than the TSK80. The 32-bit TSK3000 was 8 times faster than the custom core however the value of this extra performance would need to be assessed in the context of the requirements of the final application.

The test application was an Adaptive PCM encoder/decoder routine. The custom processor core proved it was able to encode and decode 256 samples in just over 41 ms if clocked at the modest speed of 10MHz. At this rate, voice data sampled at over 6.2 kHz could be encoded and decoded in real time making the custom core ideal for telephony applications that require an analogue bandwidth up to 3kHz.

Discussion and Concluding Remarks

The purpose of this chapter was to demonstrate how the Compiler Directed Codesign methodology can be used to iteratively converge on a processor instruction set. To this end, a *typical* embedded application was selected as a test case. This application, an ADPCM encoder/decoder program was modified slightly to make it better suited for testing and compiled using the Compiler Directed Codesign apparatus. After 20 *insn* reduction iterations, a final instruction set was determined through the assistance of a Quality Factor that balanced the benefits of a reduction in the unique *insn* count with the increase in the total *insn* count. The chosen instruction set was then used as the basis of creating a complete processor core that could run inside an FPGA and compared against other commercially available processor cores. The comparison revealed that even without extensive optimisation efforts, the custom processor developed using the Compiler Directed Codesign methodology used less FPGA resources, required less program memory than the other processors, and could execute the test application faster than all but the 32-bit processor.

The next chapter concludes this thesis by discussing its contributions, limitations and the opportunities for further work.

Chapter 7.

CONCLUSION

Summary of this Dissertation

The line of enquiry that motivated this research was one of how designers of tomorrow might capitalize on the emerging technologies of today. In general, I have been interested in the specific area of hardware/software codesign and have examined how the use of reconfigurable technologies such as FPGAs might alter the way that embedded systems might be developed in the future. In particular I have explored the extent to which compilers could be used in the codesign process.

Compilers sit at the boundary between an application's description (written in a humanly readable, high-level programming language) and the hardware platform on which the application will execute (such as a microprocessor). It is at this point that all of the requirements of the software are fully described along with a complete expression of the processing platform's capabilities. In fact it could well be argued that compilers already perform a hardware/software codesign function in that they map the software description onto the available hardware resource. Following on from this logic, I formed the hypothesis that compilers can actually be used to inform and assist designers as they attempt to solve the hardware software codesign problem in embedded systems.

Programmable Logic holds the promise of a totally configurable hardware platform that can be designed to suite any given application. But the reality is that without design tools that are capable of assisting designers to capitalize on this promise, the design freedom offered by FPGAs will not be fully realized.

Designers of FPGA-based embedded systems will be keenly aware of the opportunity to execute application functionality in hardware as opposed to software however making the decision as to which portions of the application should execute as hardware or as software is

not as straight forward as many would hope. In a review of the available literature through chapters 2 and 3, I discussed how alternate research approaches have sought to solve the hardware/software codesign challenge and concluded that regardless of the approach used, compilers are central to their success.

In Chapter 4, I outlined the foundations of a Compiler Directed Codesign platform that could be used to test the original hypothesis. The GNU Compiler Collection (GCC) was proposed as a suitable compiler to base the platform on and I presented the various attributes of this compiler that led me to this conclusion. Its limitations were also discussed along with ways to curtail their impact.

Having established a clear direction for the requirements of a Compiler Directed Codesign platform, the construction of that system was discussed in Chapter 5. A mechanism was developed to quickly retarget the compiler to a new architecture along with a means of determining the relative performance that the architecture would be able to achieve. Of particular note was the way that profiling information could be utilized to calculate dynamic performance metrics even though the target processor was yet to be implemented. The limitations of this profiling approach were discussed along with modifications that could be made to the target application to alleviate their impact.

Chapter 6 demonstrated the compiler-directed codesign methodology at work using an adaptive PCM encoder/decoder application as a test case. The results demonstrated that it was not only possible to use this technique to determine the performance impact that various processor resources or instructions might have, but that this information could then be utilized to develop a processor specifically designed around a target application. A simple quality factor was then applied to determine the ideal instruction-set architecture for the ADPCM test program.

To further verify this methodology, a complete implementation of the chosen instruction-set architecture was created and tested. Through the use of simulation, the dynamic instruction predictions that were described in Chapter 5 were validated with 100% accuracy. A comparison of the synthesized custom core with three other commercially available processor cores revealed results that were generally in favour of the custom core.

This work contributes to the body of knowledge around hardware/software codesign by demonstrating the way in which a compiler can be used to direct the partitioning process. Of note is the novel use of profiling information to calculate accurate dynamic performance predictions as well as the ability to objectively measure various instruction set architecture iterations. When combined together, the techniques described demonstrate how compiler-directed codesign can offer an insight into hardware/software tradeoffs and allow system designers to reliably converge on a solution.

Conclusions

Having set out to explore the role of compilers in hardware/software codesign and having tested the hypothesis that compilers can be used to inform and assist designers as they attempt to solve the hardware software codesign problem in embedded systems, I can now draw the following conclusions from this line of investigation:

1. **Compilers can be a useful tool in directing the development of a processor core that is designed to run a specific application.**

Whilst the traditional role of a compiler is for translating a high-level description of an application into low-level instructions of a *known* target processor, its value can be extended to encompass codesign activities as well. Compilers sit at the pivotal point between hardware and software and are therefore well positioned to inform hardware/software codesign endeavours. By including the compiler in the codesign process and by utilizing intermediate information extracted from it during compilation of the target application, designers can take a *compiler's view* of the application and objectively measure the impact that including or excluding various hardware resources will have on the performance of the application.

2. **The use of a compiler in the codesign process allows dynamic performance metrics of each of the hardware/software solutions to be accurately predicted and compared.**

Because the compiler is rebuilt at each design iteration to take into account the available hardware resource, the output of the compiler at each stage will be a true and working opcode representation of the target application for the current hardware iteration. By using this compiled output in conjunction with profiling information taken from a *real* processor, it

is possible to provide 100% accurate performance predictions for each hardware/software solution. This provides designers with objective measurements that can clearly inform the codesign process and provides designers with greater confidence in the codesign analysis.

3. **The Compiler Directed Codesign methodology can be utilized to develop a processor core that consumes fewer FPGA resources and performs better than corresponding general purpose FPGA processor cores.**

Because the Compiler Directed Codesign process makes use of a compiled view of the application as part of the codesign process, it only includes instructions in the target processor that are actually used by the compiler. This allows designers to build high-performance processor cores that are specifically designed to support individual applications without the baggage of unnecessary instructions. The reduced processor complexity may lead to a reduced FPGA resource footprint without reducing to greatly the application's performance. Furthermore, by making hardware resources available in the form of instructions to the entire application, the compiler can make use of those instructions across the entire application leading to more global optimisation opportunities.

Limitations of Work Presented

The goal of this research was to investigate the usefulness of using a compiler during the hardware/software codesign of an embedded system. Whilst it has been demonstrated that compilers can be used to direct the codesign process, it is also evident that there are a number of limitations in the methodology as it currently stands.

The methodology that has been proposed is a long way from being fully automated and still requires close supervision from a skilled developer. In time, it is hoped that a set of rules and/or heuristics could be developed that would enable the process of iterative *insn* reductions to be automated.

The base processor model that was used as part of the case study assumed the processor had access to 32, 8-bit general purpose registers and, due to the nature of how GCC has been structured, this information was essentially fixed for all iterations. Ideally a parameterised processor model would be developed so that the effect of different register quantities and sizes can also be explored.

The methodology that has been proposed is able to very accurately predict the total number of instructions that will be executed for a given application. This information can be used as

an indicator of the relative performance that each iteration of the processor model will be able to achieve. In the chosen case study, however, not all instructions were of the same length and did not execute in the same amount of time. Furthermore, the execution time of some instructions were dependent on the size of data they were operating on and so the execution time even varied within a single instruction. So whilst the total number of instructions can be accurately predicted, the exact run time can not.

Aside from the technical limitations of this work, there exists another limitation that may prohibit the commercial viability of this proposal. Chapter 1 mentioned that the high-volume nature of many embedded applications made it worthwhile to expend resources on the optimisation process however as FPGA technologies continue to mature there is a steady decline in their cost and rise in their capabilities. In this environment, the savings that can be realised by expending optimisation efforts is diminished. If there is little discernable cost difference between a system that is optimised versus one that isn't then designers are likely to focus their resources into other aspects of the design that will yield greater long-term savings or which will add value to the end product and increase revenues.

Further Application and Future Work

Ultimately this work could serve as the basis for a fully automated method of processor development that would enable embedded systems designers to begin development of the application using a high-level programming language (or even higher level modelling language). Based on a set of defined constraints, an optimised custom processor could be automatically produced for that specific application.

This work lays the foundation for Compiler Directed Codesign but stops a long way short of a fully automated process. There are still a great number of research opportunities that would see this methodology further refined and developed to the point where it might be commercially viable.

Analysis Across a Broader Range of Applications

So far I have only tested this methodology through to the construction of a processor using a single test application. The ADPCM application appeared to benefit from the Compiler Directed Codesign methodology however it can not be concluded from this single test that all applications will benefit. In order for this methodology to gain broader acceptance, it would be necessary to trial and characterize it across a greater set of test applications.

Automated Instruction Merging

One of the reasons stated for selecting GCC as the basis for the Compiler Directed Codesign platform was because it offered the ability to split and merge instruction sequences. Whilst effective use of its splitting capabilities has been utilized, only limited use of its merging capabilities has been used. It would be most interesting to explore whether an automated process could be developed that would *walk* through an *insn* tree and identify common *insn sequences*. The most common sequences could then be compressed into single instructions. A cost/benefit analysis of the merits of including such instructions could then be performed to see whether further processor efficiencies could be extracted.

One reason why this particularly intrigues me is that it could rebuild a case for custom CISC based processors. Of course this is only conjecture at this stage but it would nevertheless be an interesting line of questioning.

Exploration of the Effects of Different Optimisation Approaches

A mixture of optimisation strategies were used throughout the iterative refinement of the custom processor in Chapter 6 and this was based heavily on the author's expert knowledge of the Compiler Directed Codesign process. Providing an automated process that is less reliant on the designer's experience would provide more consistent results and be less dependent on the operator. In addition to this, an automated system should also be able to identify which optimisation approaches yield the best performance and under what circumstances. If a mixed strategy is best, then the automated system should know how to choose which strategy is best at each iteration. The effect of choosing one optimisation strategy over another needs to be further investigated and characterised.

Further Analysis of the Resource Requirements of Each Instruction

The implementation of the Compiler Directed Codesign system in Chapter 6 assumed that all *insns* were of equal complexity and duration. In reality this is unlikely to be true. A more detailed analysis is required to determine the impact on FPGA resources that each instruction has as some instructions may use a subset of existing resources whereas other instructions may gain significant performance improvements with only minor additional resources required.

This analysis would not only cover the raw FPGA resources that each instruction requires but also the impact on power and maximum clock frequency that they have. This would

make way for more optimisation strategies to be explored and for processors to be designed according to a greater set of constraints.

Automation of the *Insn* Reduction Iteration Process.

The *insn* reduction process currently requires the supervision of an expert operator who can manually control it. This is not ideal as it means that the success of the process is dependant on the skill of the operator. In the very least, the system should be able to offer advice to the user as to which *insns* should be reduced or expanded at each iteration. Ideally, however, the entire process would be automated and able to follow an optimisation strategy through to completion. Once all iterations have been exhausted, the automated system should be capable of identifying the optimal result based on a criterion.

Further Parameterisation of a Custom Processor

The Compiler Directed Codesign platform as it currently stands is somewhat limited in its ability to accommodate varying instruction word sizes along with the number and type of registers within the processor. This limits the exploration of the design space and may mask additional optimisation opportunities. Additional work should be expended to decouple these limitations from the compilation flow and thereby offer greater flexibility to the designer. This would not be an easy task as it requires significant knowledge and modification of the GCC environment.

Automated Assembler Generation

The assembler used as part of the work undertaken in Chapter 6 was constructed using a Perl script. Creating and maintaining this program was a very labour intensive process that consumed a large amount of time. While it is difficult to imagine how this could have been better automated, the pain of creating an assembler in this fashion certainly motivates me to want to find a better way. The structure of opcodes and operands is entirely dependant on the chosen instruction set architecture and so it is hard to de-couple the creation of the assembler from the creation of the target processor. Having said this though, it would be extremely beneficial to find a way to automate the assembler creation process.

Contribution of this Thesis

Having investigated the role that compilers can play in the development of custom processors, the conclusion of this thesis is that there is merit in using a Compiler to Direct the Codesign of Embedded Systems for the following reasons:

1. It provides insights into hardware/software optimisation opportunities that are difficult or impossible to derive in other ways;
2. It provides objective measurements of the impact that various hardware/software tradeoffs will have on the performance of an application, and
3. It allows designers to converge to a hardware/software solution relatively quickly and with a high degree of confidence.

REFERENCES

[1] M. Barr, "Embedded Systems Glossary," see http://www.netrino.com/Publications/Glossary/E.html, 2008.

[2] M. V. Wilkes, "Computing perspectives: the rise of the VLSI processor," *Commun. ACM,* vol. 33, pp. 16-ff., 1990.

[3] G. E. Moore, "The microprocessor: engine of the technology revolution," *Commun. ACM,* vol. 40, pp. 112-114, 1997.

[4] W. Warner, "Great Moments in microprocessor history," see http://www.ibm.com/developerworks/library/pa-microhist.html, 2008

[5] C. H. Museum, "An Overview of the History of the Software Industry," see http://www.softwarehistory.org/history/Default.htm,

[6] J. Bayko, "Great Microprocessors of the Past and Present (V 11.7.0)," see http://bwrc.eecs.berkeley.edu/CIC/archive/cpu_history.html#Sec1Part1,

[7] Databeans, "2008 Microcontrollers," see http://www.databeans.net/reports/2008_php_files/08SemiProd_Microcontrollers.php, 2008

[8] B. G. Ryder, M. L. Soffa, and M. Burnett, "The impact of software engineering research on modern programming languages," *ACM Trans. Softw. Eng. Methodol.,* vol. 14, pp. 431-477, 2005.

[9] C. Fraser, Hanson, David, *A Retargetable C compiler: design and implementation*. Redwood City, CA: Addison-Wesley Publishing Company, 1995.

[10] Bapcha, "Xilinx vs. Altera: Which Is the Real Growth Stock?," see http://seekingalpha.com/article/90239-xilinx-vs-altera-which-is-the-real-growth-stock, 2008

[11] Xilinx, "MicroBlaze Processor Reference Guide," see http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf, 2008

[12] Altera, "Nios 3.0 CPU," see http://www.altera.com/literature/ds/ds_nios_cpu.pdf,

[13] N. Tredennick and B. Shimamoto, "The Inevitability of Reconfigurable Systems," *Queue,* vol. 1, pp. 34-43, 2003.

[14] M. Horowitz, "Reconfigurable Future: The ability to produce cheaper, more compact chips is a double-edged sword.," *Queue,* vol. 1, 2003.

[15] E. S. Raymond, *The Art of Unix Programming*: Thyrsus Enterprises, 2003.

[16] W. Wolf, "A decade of hardware/software codesign," *Computer,* vol. 36, pp. 38-43, 2003.

[17] M. López-Vallejo and J. C. López, "On the hardware-software partitioning problem: System modeling and partitioning techniques," *ACM Trans. Des. Autom. Electron. Syst.,* vol. 8, pp. 269-297, 2003.

[18] P. Gupta, "Hardware-software codesign," *Potentials, IEEE,* vol. 20, pp. 31-32, 2001.

[19] C. J. N. Coelho, Jr., D. C. D. Silva, Jr., and A. O. Fernandes, "Hardware-software codesign of embedded systems," in *Integrated Circuit Design, 1998. Proceedings. XI Brazilian Symposium on*, 1998, pp. 2-8.

[20] R. Ernst, "Codesign of embedded systems: status and trends," *Design & Test of Computers, IEEE,* vol. 15, pp. 45-54, 1998.

[21] J. I. Hidalgo and J. Lanchares, "Functional partitioning for hardware-software codesign using genetic algorithms," in *EUROMICRO 97. 'New Frontiers of Information Technology'., Proceedings of the 23rd EUROMICRO Conference*, 1997, pp. 631-638.

[22] J. Axelsson, "Hardware/software partitioning of real-time systems," in *Partitioning in Hardware-Software Codesigns, IEE Colloquium on*, 1995, pp. 5/1-5/8.

[23] R. P. Dick and N. K. Jha, "CORDS: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems," in *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, San Jose, California, United States, 1998, pp. 62--67.

[24] J.-M. Chang and M. Pedram, "Codex-dp: co-design of communicating systems using dynamic programming," in *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, 1999, pp. 568-573.

[25] P. V. Knudsen and J. Madsen, "Graph based communication analysis for hardware/software codesign," in *Hardware/Software Codesign, 1999. (CODES '99) Proceedings of the Seventh International Workshop on*, 1999, pp. 131-135.

[26] Altium, "Getting Started with the C-to-Hardware Compiler," see http://www.altium.com/files/altiumdesigner6/learningguides/TU0130%20Getting%20Started%20with%20the%20C-to-Hardware%20Compiler.pdf, 2008

[27] M. J. Knieser and C. A. Papachriston, "COMET: a hardware-software codesign methodology," in *Design Automation Conference, 1996, with EURO-VHDL '96 and Exhibition, Proceedings EURO-DAC '96, European*, 1996, pp. 178-183.

[28] M. Edwards and B. Fozard, "Rapid prototyping of mixed hardware and software systems," in *Digital System Design, 2002. Proceedings. Euromicro Symposium on*, 2002, pp. 118-125.

[29] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-Software Co-Design of Embedded Reconfigurable Architectures," in *DAC 2000*, Los Angeles, CA USA, 2000, pp. 507-512.

[30] R. Schreiber, S. Aditya, B. Ramakrishna Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider, "High-level synthesis of nonprogrammable hardware accelerators," in *Application-Specific Systems, Architectures, and Processors, 2000. Proceedings. IEEE International Conference on*, 2000, pp. 113-124.

[31] L. Theriault, D. Auder, and Y. Savaria, "Performance estimators for hardware/software co-design," in *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, 2001, pp. 17-20 vol. 5.

[32] G. Stitt, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: a first approach," in *Proceedings of the 40th conference on Design automation*, Anaheim, CA, 2003, pp. 250-255.

[33] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, and A. Sangiovanni-Vincentelli, "HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform," in *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*, 2002, pp. 151-156.

[34] A. Y. Alomary, "A hardware/software codesign partitioner for ASIP design," in *Electronics, Circuits, and Systems, 1996. ICECS '96., Proceedings of the Third IEEE International Conference on*, 1996, pp. 251-254 vol.1.

[35] I.-J. Huang and A. M. Despain, "Generating instruction sets and microarchitectures from applications," in *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, San Jose, California, United States, 1994, pp. 391--396.

[36] M. Breternitz, Jr. and J. P. Shen, "Architecture synthesis of high-performance application-specific processors," in *Design Automation Conference, 1990. Proceedings. 27th ACM/IEEE*, 1990, pp. 542-548.

[37] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An Instruction Set Description Language for Retargetability," in *DAC'97*, Anaheim, CA, USA, 1997.

[38] A. Fauth, J. Van Praet, and M. Freericks, "Describing Instruction Set Processors Using nML," pp. 503-507, 1995 1995.

[39] J.-e. Lee, K. Choi, and N. Dutt, "Efficient instruction encoding for automatic instruction set design of configurable ASIPs," in *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, San Jose, California, 2002, pp. 649-654.

[40] Q. Zhao, B. Mesman, and T. Basten, "Practical Instruction Set Design and Compiler Retargetability Using Static Resource Models," in *Proceedings of the conference on Design, automation and test in Europe*, 2002, p. 1021.

[41] B. K. Holmer, "A tool for processor instruction set design," in *Proceedings of the conference on European design automation*, Grenoble, France, 1994, pp. 150-155.

[42] C. Liem, P. Paulin, and A. Jerraya, "ReCode: the design and re-design of the instruction codes for embedded instruction-set processors," in *European Design and Test Conference, 1997. ED&TC 97. Proceedings*, 1997, p. 612.

[43] R. E. Gonzalez, "Xtensa: a configurable and extensible processor," *Micro, IEEE,* vol. 20, pp. 60-70, 2000.

[44] A. Wang, E. Killian, D. Maydan, and C. Rowen, "Hardware/software instruction set configurability for system-on-chip processors," in *Design Automation Conference, 2001. Proceedings*, 2001, pp. 184-188.

[45] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems* San Jose, California, USA: ACM, 2003, pp. 137-147.

[46] A. Alomary, T. Nakata, Y. Honma, J. Sato, N. Hikichi, and M. Imai, "PEAS-I: A hardware/software co-design system for ASIPs," in *Design Automation Conference, 1993, with EURO-VHDL '93. Proceedings EURO-DAC '93. European*, 1993, pp. 2-7.

[47] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai, "PEAS-III: an ASIP design environment," in *Computer Design, 2000. Proceedings. 2000 International Conference on*, 2000, pp. 430-436.

[48] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proceeding of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, Monterey, California, USA, 2004, pp. 183-189.

[49] B. F. Veale, J. K. Antonio, M. P. Tull, and S. A. Jones, "Selection of instruction set extensions for an FPGA embedded processor core," in *20th International Parallel and Distributed Processing Symposium, 2006. IPDPS 2006.*, 2006, p. 8 pp.

[50] P. Bonzini and L. Pozzi, "Code transformation strategies for extensible embedded processors," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems* Seoul, Korea: ACM, 2006, pp. 242-252.

[51] U. Kastens, D. Khoi Le, A. Slowik, and M. Thies, "Feedback driven instruction-set extension," in *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, Washington, DC, USA, 2004, pp. 126-135.

[52] J. Peddersen, S. L. Shee, A. Janapsatya, and S. Parameswaran, "Rapid Embedded Hardware/Software System Generation," in *Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*: IEEE Computer Society, 2005, pp. 111-116.

[53] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami, "Determining the optimum extended instruction-set architecture for application specific reconfigurable VLIW CPUs," in *Rapid System Prototyping, 12th International Workshop on, 2001.*, 2001, pp. 50-56.

[54] M. Boden, J. Schneider, K. Feske, and S. Rulke, "Enhanced reusability for SoC-based HW/SW co-design," in *Digital System Design, 2002. Proceedings. Euromicro Symposium on*, 2002, pp. 94-99.

[55] F. Barat, R. Lauwereins, and G. Deconinck, "Reconfigurable instruction set processors from a hardware/software perspective," *Software Engineering, IEEE Transactions on,* vol. 28, pp. 847-862, 2002.

[56] L. A. Belady, R. A. Nelson, and G. S. Shedler, "An anomaly in space-time characteristics of certain programs running in a paging machine," *Commun. ACM,* vol. 12, pp. 349-353, 1969.

[57] M. K. Jain, L. Wehmeyer, S. Steinke, P. Marwedel, and M. Balakrishnan, "Evaluating register file size in ASIP design," in *Proceedings of the ninth international symposium on Hardware/software codesign* Copenhagen, Denmark: ACM, 2001, pp. 109-114.

[58] D. G. Bradlee, S. J. Eggers, and R. R. Henry, "The Effect on RISC Performance of Register Set Size and Structure Versus Code Generation Strategy," in *Proc. 18th Annual Symposium on Computer Architecture*, 1991, pp. 330-339.

[59] G. M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities.," Atlantic City, New Jersey, 1967, pp. 483-485.

[60] J. G. D'Ambrosio and X. S. Hu, "Configuration-Level Hardware/ Software Partitioning for Real-Time Embedded Systems," in *Proceedings of the 3rd international workshop on Hardware/software co-design*, Grenoble, France, 1994, pp. 34-41.

[61] D. Ragan, P. Sandborn, and P. Stoaks, "A Detailed Cost Model for Concurrent Use With Hardware/Software Co-Design," in *Design Automation Conference 2002*, New Orleans, Louisiana, USA, 2002, pp. 269-274.

[62] L. Carro, G. A. Pereira, C. Alba, and A. Suzim, "System design using ASIPs," in *Engineering of Computer-Based Systems,1996. Proceedings., IEEE Symposium and Workshop on*, 1996, pp. 80-85.

[63] M. D. Edwards and J. Forrest, "Hardware/software partitioning for performance enhancement," in *Partitioning in Hardware-Software Codesigns, IEE Colloquium on*, 1995, pp. 2/1-2/5.

[64] J. Henkel, "A low power hardware/software partitioning approach for core-based embedded systems," in *Design Automation Conference, 1999. Proceedings. 36th*, 1999, pp. 122-127.

[65] J. Noguera and R. M. Badia, "System-level power-performance trade-offs in task scheduling for dynamically reconfigurable architectures," in *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems*, San Jose, California, 2003, pp. 73-83.

[66] M. Meerwein, C. Baumgartner, and W. Gauert, "Linking codesign and reuse in embedded systems design," in *Hardware/Software Codesign, 2000. CODES 2000. Proceedings of the Eighth International Workshop on*, San Diego, CA, USA, 2000, pp. 93-97.

[67] B. Shackleford, M. Yasuda, E. Okushi, H. Koizumi, H. Tomiyama, and H. Yasuura, "Memory-CPU size optimization for embedded system designs," in *DAC '97: Proceedings of the 34th annual conference on Design automation*, Anaheim, California, United States, 1997, pp. 246--251.

[68] Y. Le Moullec, J. P. Diguet, and J. L. Philippe, "A scheduling framework for system-level estimation," in *Electronics, Circuits and Systems, 2000. ICECS 2000. The 7th IEEE International Conference on*, 2000, pp. 277-280 vol.1.

[69] R. K. Gupta and G. D. Micheli, "Constrained software generation for hardware-software systems," in *Proceedings of the 3rd international workshop on Hardware/software co-design* Grenoble, France: IEEE Computer Society Press, 1994, pp. 56-63.

[70] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "Source-level execution time estimation of C programs," in *Proceedings of the ninth international symposium on Hardware/software codesign*, Copenhagen, Denmark, 2001, pp. 98-103.

[71] J. A. Maestro, D. Mozos, and H. Mecha, "A macroscopic time and cost estimation model allowing task parallelism and hardware sharing of the codesign partitioning process," in *Proceedings of the conference on Design, automation and test in Europe*, Le Palais des Congrés de Paris, France, 1998, pp. 218-225.

[72] P. V. Knudsen and J. Madsen, "PACE: A Dynamic Programming Algorithm for Hardware/Software Partitioning," in *Proceedings of the 4th International Workshop on Hardware/Software Co-Design*, 1996, p. 85.

[73] "Knapsack problem," see http://en.wikipedia.org/wiki/Knapsack_problem, 2007

[74] N. Cheung, J. Henkel, and S. Parameswaran, "Rapid configuration and instruction selection for an ASIP: a case study," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003, pp. 802-807.

[75] J. Henkel and R. Ernst, "An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* vol. 9, pp. 273-289, 2001.

[76] W. Fornaciari, P. Gubian, D. Sciuto, and C. Silvano, "Power estimation of embedded systems: a hardware/software codesign approach," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* vol. 6, pp. 266-275, 1998.

[77] H. Mizuno, H. Kobayashi, T. Onoye, and I. Shirakawa, "Power estimation at architecture level for embedded systems," in *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*, 2002, pp. II-476-II-479 vol.2.

[78] J. P. Brage and J. Madsen, "A codesign case study in computer graphics," in *Hardware/Software Codesign, 1994., Proceedings of the Third International Workshop on*, 1994, pp. 132-139.

[79] J. T. Russell and M. F. Jacome, "Scenario-based software characterization as a contingency to traditional program profiling," in *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, Grenoble, France, 2002, pp. 170--177.

[80] D. Bjorklund and J. Lilius, "A language for multiple models of computation," in *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*, 2002, pp. 25-30.

[81] C. A. M. Marcon, N. L. V. Calazans, and F. G. Moraes, "Requirements, primitives and models for systems specification," in *Integrated Circuits and Systems Design, 2002. Proceedings. 15th Symposium on*, 2002, pp. 323-328.

[82] C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng, "Overview of the Ptolemy Project," University of California, Dep. of Electrical Engineering and Computer Science, California July 2, 2003 2003.

[83] S. J. Cunning, T. C. Ewing, J. T. Olson, J. W. Rozenblit, and S. Schulz, "Towards an integrated, model-based codesign environment," in *Engineering of Computer-Based Systems, 1999. Proceedings. ECBS '99. IEEE Conference and Workshop on*, 1999, pp. 136-143.

[84] S. Schulz, J. W. Rozenblit, M. Mrva, and K. Buchenriede, "Model-based codesign," *IEEE Computer,* vol. 31, pp. 60-67, 1998.

[85] P. Green, P. Rushton, and R. Beggs, "An example of applying the codesign method MOOSE," in *Hardware/Software Codesign, 1994., Proceedings of the Third International Workshop on*, 1994, pp. 65-72.

[86] F. Muller, J. P. Calvez, D. Heller, and O. Pasquier, "An interactive modeling and generation tool for the design of Hw/Sw systems," in *EUROMICRO Conference, 1999. Proceedings. 25th*, 1999, pp. 382-385 vol.1.

[87] V. Carchiolo, M. Malgeri, and G. Mangioni, "TTL: a modular language for hardware/software systems design," *Journal of Computer and System Sciences,* vol. 66, pp. 293-315, 2003/3 2003.

[88] V. Carchiolo, M. Malgeri, and G. Mangioni, "Hardware/software synthesis of formal specifications in codesign of embedded systems," *ACM Trans. Des. Autom. Electron. Syst.,* vol. 5, pp. 399-432, 2000.

[89] T. B. Ismail and A. A. Jerraya, "Design models and steps for codesign," in *Verification of Hardware Software Codesign, IEE Colloquium on*, 1995, pp. 1/1-1/2.

[90] A. Muth and G. Farber, "SDL as a system level specification language for application-specific hardware in a rapid prototyping environment," in *System Synthesis, 2000. Proceedings. The 13th International Symposium on*, 2000, pp. 157-162.

[91] F. Slomka, M. Dorfel, and R. Munzenberger, "Generating Mixed Hardware/Software Systems from SDL Specifications," in *CODES'01*, Copenhagen, Denmark, 2001, pp. 116-121.

[92] R. Braek and A. Meisingset, "The ITU-T Languages in a Nutshell," *Telektronikk,* p. 19, 2000.

[93] J.-M. Daveau, G. F. Marchioro, C. A. Valderrama, and A. A. Jerraya, "VHDL generation from SDL specifications," in *Proceedings of the IFIP TC10 WG10.5 international conference on Hardware description languages and their applications*, Toledo, Spain, 1997, pp. 182-201.

[94] C. A. M. Marcon, F. Hessel, A. M. Amory, L. H. L. Ries, F. G. Moraes, and N. L. V. Calazans, "Prototyping of embedded digital systems from SDL language: a case study," in *High-Level Design Validation and Test Workshop, 2002. Seventh IEEE International*, 2002, pp. 133-138.

[95] "The ESTEREL Language," see http://www-sop.inria.fr/meije/esterel/esterel-eng.html,

[96] F. Balarin, M. Chiodo, D. W. Engels, P. Giusto, W. Gosti, H. Hsieh, A. Jurecska, M. Lajolo, L. Lavagno, C. Passerone, R. Passerone, C. Sansoe, M. Sgroi, E. Sentovich, K. Suzuki, B. Tabbara, R. von Hanxleden, S. Yee, and A. Sangiovanni-Vincentelli, "Polis - A design environment for control-dominated embedded systems," Berkeley 10/11/1999 1999.

[97] L. Lavagno, M. Chiodo, P. Giusto, A. Jurecska, H. Hsieh, S. Yee, A. S. Sangiovanni-Vincentelli, and R. Suzuki, "A case study in computer-aided codesign of embedded controllers," in *Hardware/Software Codesign, 1994., Proceedings of the Third International Workshop on*, 1994, pp. 220-224.

[98] F. Balarin and M. Chiodo, "Software synthesis for complex reactive embedded systems," in *Computer Design, 1999. (ICCD '99) International Conference on*, 1999, pp. 634-639.

[99] C. Stoel and J. Karrfalt, "VIOOL for hardware/software codesign," in *Systems Engineering of Computer Based Systems, 1995., Proceedings of the 1995 International Symposium and Workshop on*, 1995, pp. 333-340.

[100] E. Stoy and Z. Peng, "An integrated modelling technique for hardware/software systems," in *Circuits and Systems, 1994. ISCAS '94., 1994 IEEE International Symposium on*, 1994, pp. 399-402 vol.1.

[101] K. Ramani and R. L. Haggard, "A survey of techniques used in the synthesis of hardware from C/C++ as a part of hardware/software co-design," in *Southeastern Symposium on System Theory, 2001. Proceedings of the 33rd*, 2001, pp. 301-304.

[102] S. Sankaran and R. L. Haggard, "A convenient methodology for efficient translation of C to VHDL," in *Southeastern Symposium on System Theory, 2001. Proceedings of the 33rd*, 2001, pp. 203-207.

[103] V. V. Sanevelly and R. L. Haggard, "A procedure for designing a translator from C to VHDL," in *System Theory, 2002. Proceedings of the Thirty-Fourth Southeastern Symposium on*, 2002, pp. 329-333.
[104] Celoxica, "Handel-C for Hardware Design," see http://www.celoxica.com/techlib/files/CEL-W0307171L48-63.pdf,
[105] IEEE, "IEEE Standard SystemC Language Reference Manual," see http://standards.ieee.org/getieee/1666/download/1666-2005.pdf, 2007
[106] W. Ecker, "Using VHDL for HW/SW co-specification," in *Design Automation Conference, 1993, with EURO-VHDL '93. Proceedings EURO-DAC '93. European*, 1993, pp. 500-505.
[107] I. Page, "Constructing hardware-software systems reliably from a single description," in *Structured Methods for Hardware Systems Design, IEE Colloquium on*, 1994, pp. 2/1-2/9.
[108] G. Barrett, "Occam 3 Reference Manual," INMOS Limited, 1992.
[109] I. L. Taylor, "New target porting time estimates?," see http://www.cygwin.com/ml/binutils/2003-11/msg00235.html,
[110] O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr, "Architecture implementation using the machine description language LISA," in *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings.*, 2002, pp. 239-244.
[111] S. Hanono and S. Devadas, "Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator," in *Proceedings of the 35th annual conference on Design automation*, San Francisco, California, United States, 1998, pp. 510--515.
[112] M. Gschwind, "Instruction set selection for ASIP design," in *Hardware/Software Codesign, 1999. (CODES '99) Proceedings of the Seventh International Workshop on*, 1999, pp. 7-11.
[113] O. Hebert, I. C. Kraljic, and Y. Savaria, "A Method To Derive Application-Specific Embedded Processing Cores," in *CODES 2000*, San Diego, CA USA, 2000.
[114] P. Yu and T. Mitra, "Characterizing embedded applications for instruction-set extensible processors," in *Proceedings of the 41st annual conference on Design automation*, San Diego, CA, USA, 2004, pp. 723-728.
[115] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Design Automation Conference, 2003. Proceedings*, 2003, pp. 256-261.
[116] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Synthesis of custom processors based on extensible platforms," in *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, San Jose, California, 2002, pp. 641-648.
[117] D. Fischer, J. Teich, M. Thies, and R. Weper, "Efficient architecture/compiler co-exploration for ASIPs," in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems CASES2002*, Grenoble, France, 2002, pp. 27-34.
[118] M. Imai, J. Sato, A. Alomary, and N. Hikichi, "An integer programming approach to instruction implementation method selection problem," in *EURO-DAC '92: Proceedings of the conference on European design automation*, Congress Centrum Hamburg, Hamburg, Germany, 1992, pp. 106--111.
[119] N. N. Binh, M. Imai, A. Shiomi, and N. Hikichi, "A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate counts," in *DAC '96: Proceedings of the 33rd annual conference on Design automation*, Las Vegas, Nevada, United States, 1996, pp. 527--532.

[120] J. Sato, M. Imai, T. Hakata, A. Y. Alomary, and N. Hikichi, "An Integrated Design Environment for Application Specific Integrated Processor," in *ICCD '91: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer \& Processors*, 1991, pp. 414--417.
[121] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi, "An ASIP instruction set optimization algorithm with functional module sharing constraint," in *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, Santa Clara, California, United States, 1993, pp. 526--532.
[122] Tensilica, "Automated Configurable Processor Design Flow," Tensilica, 2005.
[123] Xilinx, "MicroBlaze, An enhanced 32-Bit Processor Core for FPGA Integration," see http://ramp.eecs.berkeley.edu/Publications/MBforRAMP2.ppt, 2008
[124] Tensilica, "Xtensa-V Configurable Processor," see, 2008
[125] Xilinx, "MicroBlaze Processor Performance," see http://www.xilinx.com/products/design_resources/proc_central/microblaze_per.htm, 2008
[126] "Free Pascal," see http://www.freepascal.org/,
[127] "GNU Pascal," see http://www.gnu-pascal.de/gpc/h-index.html,
[128] "GNU Compiler Collection - GCC," see http://gcc.gnu.org,
[129] "Open Watcom," see http://www.openwatcom.org/index.php/Main_Page,
[130] "SDCC - Small Device C Compiler," see http://sdcc.sourceforge.net/,
[131] "OpenCOBOL," see http://www.opencobol.org/,
[132] "Viva - Open Source Java," see http://viva.sourceforge.net/,
[133] "The Pizza Compiler," see http://pizzacompiler.sourceforge.net/,
[134] Wikipedia, "GNU Compiler Collection," see http://en.wikipedia.org/wiki/GNU_Compiler_Collection, 2005
[135] R. M. Stallman, *GNU Compiler Collection Internals*, for GCC 3.4 ed.: Free Software Foundation, 2002.
[136] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. R. Brown, "MiBench: A free, commercially representative embedded benchmark," in *IEEE 4th Annual Workshop on Workload Characterization*, Austin, Texas, 2001.
[137] "MiBench Version 1.0," http://www.eecs.umich.edu/mibench/: University of Michigan, 2001.
[138] "ModelSim," ModelSim XE II 5.7c ed: Mentor Graphics, 2005.
[139] "Altium Designer," Altium Designer 6 ed: Altium, 2006.
[140] Altium, "CR0115 TSK51x MCU," see http://www.altium.com/files/altiumdesigner/s08/learningguides/CR0115%20TSK51x%20MCU.pdf, 2008
[141] Altium, "CR0117 TSK80x MCU," see http://www.altium.com/files/altiumdesigner/s08/learningguides/CR0117%20TSK80x%20MCU.pdf, 2008
[142] Altium, "CR0121 TSK3000A 32-bit RISC Processor," see http://www.altium.com/files/altiumdesigner/s08/learningguides/CR0121%20TSK3000A%2032%20bit%20RISC%20Processor.pdf, 2008
[143] "FPGA Processors Resource Usage," Altium Limited 1/8/2007 2006.
[144] "FPGA Peripheral Resource Usage," Altium Limited 1/8/2006 2005.

APPENDIX A: CODE LISTING OF ADPCM

```
/*
** ADPCM - Test timing on adpcm coder and decoder.
**
** The program runs the ADPCM compressor and decompressor on a reference dataset
** and compares the result with a second reference dataset to determine any
** processing errors.
*/

#define DATASIZE 256 /* Data block size */

typedef char word8;
typedef unsigned char uword8;
typedef short word16;
typedef unsigned short uword16;
typedef int word32;
typedef unsigned int uword32;

word16 __time = 0;


struct adpcm_state {
    word16 valprev; /* Previous output value */
    word8 index;   /* Index into stepsize table */
};

void adpcm_coder(word8 indata[], word8 outdata[], word16 len, struct adpcm_state*
state);
void adpcm_decoder(word8 indata[], word8 outdata[], word16 len, struct adpcm_state*
state);

word8 RefPCMdata[] =
{
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38,
0, 38, 70, 92, 99, 92, 70, 38, 0, -38, -70, -92, -100, -92, -70, -38
};

word8 RefPCMdata_2[] =
{
0, 11, 41, 96, 103, 97, 66, 39, 4, -37, -75, -90, -103, -91, -73, -35,
0, 41, 68, 93, 97, 93, 68, 37, -1, -36, -67, -96, -99, -89, -68, -38,
0, 35, 66, 95, 98, 95, 67, 42, -3, -34, -72, -97, -101, -89, -71, -40,
-2, 33, 74, 90, 95, 91, 70, 36, 5, -41, -72, -88, -103, -90, -69, -35,
-4, 42, 73, 89, 104, 91, 70, 36, 5, -41, -72, -88, -103, -90, -69, -35,
-4, 42, 73, 89, 104, 91, 70, 36, 5, -41, -72, -88, -103, -90, -69, -35,
-4, 42, 73, 89, 104, 91, 70, 36, 5, -41, -72, -88, -103, -90, -69, -35,
-4, 42, 73, 89, 104, 91, 70, 36, 5, -41, -72, -88, -103, -90, -69, -35,
-4, 42, 73, 89, 104, 91, 70, 36, 5, -41, -72, -88, -103, -90, -69, -35,
-4, 42, 73, 89, 104, 91, 70, 36, 5, -41, -72, -88, -103, -90, -69, -35,
-4, 42, 73, 89, 104, 91, 70, 36, 5, -41, -72, -88, -103, -90, -69, -35,
-4, 42, 73, 89, 104, 91, 70, 36, 5, -41, -72, -88, -103, -90, -69, -35,
-4, 42, 73, 89, 104, 91, 70, 36, 5, -41, -72, -88, -103, -90, -69, -35,
-4, 42, 73, 89, 104, 91, 70, 36, 5, -41, -72, -88, -103, -90, -69, -35,
-4, 42, 73, 89, 104, 91, 70, 36, 5, -41, -72, -88, -103, -90, -69, -35
```

```
};

word8 adpcmdata[DATASIZE / 2];
word8 pcmdata_2[DATASIZE];

struct adpcm_state coder_1_state, coder_2_state, decoder_state;

/* Intel ADPCM step variation table */
static word8 indexTable[16] =
{
    - 1, - 1, - 1, - 1, 2, 4, 6, 8, - 1, - 1, - 1, - 1, 2, 4, 6, 8,
};

static word16 stepsizeTable[89] =
{
7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 19, 21, 23, 25, 28, 31, 34, 37, 41, 45, 50, 55, 60,
66, 73, 80, 88, 97, 107, 118, 130, 143, 157, 173, 190, 209, 230, 253, 279, 307, 337,
371, 408, 449, 494, 544, 598, 658, 724, 796, 876, 963, 1060, 1166, 1282, 1411, 1552,
1707, 1878, 2066, 2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358, 5894,
6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899, 15289, 16818, 18500, 20350,
22385, 24623, 27086, 29794, 32767
};


void adpcm_coder(word8 indata[], word8 outdata[], word16 len, struct adpcm_state* state)
{
    word8       * inp; /* Input buffer pointer */
    word8       * outp; /* output buffer pointer */
    word16           val; /* Current input sample value */
    uword8      sign; /* Current adpcm sign bit */
    word16           delta; /* Current adpcm output value */
    word16           diff; /* Difference between val and valprev */
    word16           step; /* Stepsize */
    word32           valpred; /* Predicted output value */
    word16           vpdiff; /* Current change to valpred */
    word8           index; /* Current step change index */
    word8           outputbuffer; /* place to keep previous 4-bit value */
    word8           bufferstep; /* toggle between outputbuffer/output */

    outp = (word8*) outdata;
    inp = (word8*) indata;

    valpred = state->valprev;
    index = state->index;
    step = stepsizeTable[index];

    bufferstep = 1;

      //    for (; len > 0; len--)
    //    while (len > 0)
    while (1)
    {
      if (len <= 0)
            break;

        val = * inp++;

        /* Step 1 - compute difference with previous value */
        diff = val - valpred;

            //        sign = (diff < 0) ? 8: 0;
            if (diff < 0)
              {
                sign = 8;
              }
            else
              {
                sign = 0;
              }

        if (sign)
            diff = (- diff);

        delta = 0;
```

```
        vpdiff = (step >> 3);

        if (diff >= step)
        {
            delta = 4;
            diff -= step;
            vpdiff += step;
        }
        step >>= 1;
        if (diff >= step)
        {
            delta |= 2;
            diff -= step;
            vpdiff += step;
        }
        step >>= 1;
        if (diff >= step)
        {
            delta |= 1;
            vpdiff += step;
        }

        /* Step 3 - Update previous value */
        if (sign)
            valpred -= vpdiff;
        else
            valpred += vpdiff;

            /* Step 4 - Clamp previous value to 16 bits */
        if (valpred > 32767)
            valpred = 32767;
        else if (valpred < - 32768)
            valpred = - 32768;

            /* Step 5 - Assemble value, update index and step values */
        delta |= sign;

        index += indexTable[delta];
        if (index < 0)
            index = 0;
        if (index > 88)
            index = 88;
        step = stepsizeTable[index];

        /* Step 6 - Output value */
        if (bufferstep)
        {
            outputbuffer = (delta << 4) & 0xf0;
        }
        else
        {
            * outp++ = (delta & 0x0f) | outputbuffer;
        }
        bufferstep = !bufferstep;

            len--;
    }

    /* Output last step, if needed */
    if (!bufferstep)
        * outp++ = outputbuffer;

    state->valprev = (word16)valpred;
    state->index = index;
}

void adpcm_decoder(word8 indata[], word8 outdata[], word16 len, struct adpcm_state*
state)
{
    word8 * inp; /* Input buffer pointer */
    word8       * outp; /* output buffer pointer */
    uword8 sign; /* Current adpcm sign bit */
    word16           delta; /* Current adpcm output value */
    word16           step; /* Stepsize */
```

```
word32           valpred; /* Predicted value */
word16           vpdiff; /* Current change to valpred */
word8           index; /* Current step change index */
word8           inputbuffer; /* place to keep next 4-bit value */
word8           bufferstep; /* toggle between inputbuffer/input */

outp = (word8*) outdata;
inp = (word8*) indata;

valpred = state->valprev;
index = state->index;
step = stepsizeTable[index];

bufferstep = 0;

//    for (; len > 0; len--)
//while (len > 0)
while(1)
{
  if (len <= 0)
        break;

    /* Step 1 - get the delta value */
    if (bufferstep)
    {
        delta = inputbuffer & 0xf;
    }
    else
    {
        inputbuffer = * inp++;
        delta = (inputbuffer >> 4) & 0xf;
    }
    bufferstep = !bufferstep;

    /* Step 2 - Find new index value (for later) */
    index += indexTable[delta];
    if (index < 0)
        index = 0;
    if (index > 88)
        index = 88;

        /* Step 3 - Separate sign and magnitude */
    sign = delta & 8;
    delta = delta & 7;

    /* Step 4 - Compute difference and new predicted value */
    /*
    ** Computes 'vpdiff = (delta+0.5)*step/4', but see comment
    ** in adpcm_coder.
    */
    vpdiff = step >> 3;
    if (delta & 4)
        vpdiff += step;
    if (delta & 2)
        vpdiff += step >> 1;
    if (delta & 1)
        vpdiff += step >> 2;

    if (sign)
        valpred -= vpdiff;
    else
        valpred += vpdiff;

        /* Step 5 - clamp output value */
    if (valpred > 32767)
        valpred = 32767;
    else if (valpred < - 32768)
        valpred = - 32768;

        /* Step 6 - Update step value */
    step = stepsizeTable[index];

    /* Step 7 - Output value */
    * outp++ = valpred;
```

```
            len--;
    }

    state->valprev = valpred;
    state->index = index;
}

void main(void)
{
    word16 i, ErrorCount;

    adpcm_coder(RefPCMdata, adpcmdata, DATASIZE, &coder_2_state);
    adpcm_decoder(adpcmdata, pcmdata_2, DATASIZE, &decoder_state);

      //    for (i = 0, ErrorCount = 0; i < DATASIZE; i++)
    i = 0;
    ErrorCount = 0;
    //    while (i < DATASIZE)
    while (1)
      {
            if (i >= DATASIZE)
              break;

            if (pcmdata_2[i] != RefPCMdata_2[i])
              {
                ErrorCount++;
              }
            i++;
      }

    if (ErrorCount != 0)
      {
            __time = ErrorCount;
      }
    else
      {
            __time++;
      }
}
```

APPENDIX B: CONTENTS OF CD

Custom Processor Core

- VHDL Sources for the processor core developed as a result of the analysis performed in chapter 6.

Experimental Results

- GCC_Custom: Copies of files used in the generation of the custom processor machine description file.
- Scripts: Used for running each iteration of the compiler
- Snapshots: Dumps of the compiler output from all experimental results pertinent to this thesis

GCC

- Contains a compressed image of the Compiler Directed Codesign environment based on GCC 3.3 along with instructions on how to set up the environment.

INSN Assembler

- Contains the PERL script used for assembling the Compiler output into a format that can be tested on the custom processor core.

MiBench

- Full sources for the MiBench benchmarking test suite.

Thesis – Original Submission

- Thesis document as it was presented for examination

Thesis – Final

- The final thesis document with amendments based on examiners' comments